Page Contents

Astropy Documentation

astropy
A Community Python Library for Astronomy

The `astropy` package contains key functionality and common tools needed for performing astronomy and astrophysics with Python. It is at the core of the Astropy Project, which aims to enable the community to develop a robust ecosystem of affiliated packages covering a broad range of needs for astronomical research, data processing, and data analysis.

Important

If you use Astropy for work presented in a publication or talk please help the project via proper citation or acknowledgement. This also applies to use of software or affiliated packages that depend on the astropy core package.

Getting Started

Installation

Requirements

`astropy` has the following strict requirements:

- Python 3.7 or later
- Numpy 1.17.0 or later
- PyERFA 1.7 or later

`astropy` also depends on other packages for optional features:

- scipy 1.1 or later: To power a variety of features in several modules.
- h5py: To read/write **`Table`** objects from/to HDF5 files.
- BeautifulSoup: To read **`Table`** objects from HTML files.
- html5lib: To read **`Table`** objects from HTML files using the pandas reader.
- bleach: Used to sanitize text when disabling HTML escaping in the **`Table`** HTML writer.
- PyYAML 3.13 or later: To read/write **`Table`** objects from/to the Enhanced CSV ASCII table format and to serialize mixins for various formats.
- xmllint: To validate VOTABLE XML files. This is a command line tool installed outside of Python.
- pandas: To convert **`Table`** objects from/to pandas DataFrame objects. Version 0.14 or higher is required to use the Pandas I/O functions to read/write **`Table`** objects.
- sortedcontainers for faster `SCEngine` indexing engine with `Table`, although this may still be slower in some cases than the default indexing engine.
- pytz: To specify and convert between timezones.
- jplephem: To retrieve JPL ephemeris of Solar System objects.
- matplotlib 3.0 or later: To provide plotting functionality that **`astropy.visualization`** enhances.
- setuptools: Used for discovery of entry points which are used to insert fitters into **`astropy.modeling.fitting`**.
- mpmath: Used for the 'kraft-burrows-nousek' interval in **`poisson_conf_interval`**.
- asdf 2.6.0 or later: Enables the serialization of various Astropy classes into a portable, hierarchical, human-readable representation.
- bottleneck: Improves the performance of sigma-clipping and other functionality that may require computing statistics on arrays with NaN values.

However, note that these packages require installation only if those particular features are needed. `astropy` will import even if these dependencies are not installed.

The following packages can optionally be used when testing:

- pytest-astropy: See Testing a Source Code Build of astropy
- pytest-xdist: Used for distributed testing.
- pytest-mpl: Used for testing with Matplotlib figures.
- objgraph: Used only in tests to test for reference leaks.
- IPython: Used for testing the notebook interface of **`Table`**.
- coverage: Used for code coverage measurements.
- skyfield: Used for testing Solar System coordinates.
- spgp4: Used for testing satellite positions.
- tox: Used to automate testing and documentation builds.

Installing `astropy`

If you are new to Python and/or do not have familiarity with Python virtual environments, then we recommend starting by installing the Anaconda Distribution. This works on all platforms (linux, Mac, Windows) and installs a full-featured scientific Python in a user directory without requiring root permissions.

Using pip

Warning

Users of the Anaconda Python distribution should follow the instructions for Using Conda.

To install `astropy` with pip, run:

```
pip install astropy
```

If you want to make sure none of your existing dependencies get upgraded, you can also do:

```
pip install astropy --no-deps
```

On the other hand, if you want to install `astropy` along with all of the available optional dependencies, you can do:

```
pip install astropy[all]
```

In most cases, this will install a pre-compiled version (called a *wheel*) of astropy, but if you are using a very recent version of Python, if a new version of astropy has just been released, or if you are building astropy for a platform that is not common, astropy will be installed from a source file. Note that in this case you will need a C compiler (e.g., `gcc` or `clang`) to be installed (see Building from source below) for the installation to succeed.

If you get a `PermissionError` this means that you do not have the required administrative access to install new packages to your Python installation. In this case you may consider using the `--user` option to install the package into your home directory. You can read more about how to do this in the pip documentation.

Alternatively, if you intend to do development on other software that uses `astropy`, such as an affiliated package, consider installing `astropy` into a virtualenv.

Do **not** install `astropy` or other third-party packages using `sudo` unless you are fully aware of the risks.

Using Conda

To install `astropy` using conda run:

```
conda install astropy
```

`astropy` is installed by default with the Anaconda Distribution. To update to the latest version run:

```
conda update astropy
```

There may be a delay of a day or two between when a new version of `astropy` is released and when a package is available for conda. You can check for the list of available versions with `conda search astropy`.

It is highly recommended that you install all of the optional dependencies with:

```
conda install -c astropy -c defaults \
  scipy h5py beautifulsoup4 html5lib bleach pyyaml pandas
sortedcontainers \
  pytz matplotlib setuptools mpmath bottleneck jplephem asdf
```

To also be able to run tests (see below) and support Building Documentation use the following. We use `pip` for these packages to ensure getting the latest releases which are compatible with the latest `pytest` and `sphinx` releases:

```
pip install pytest-astropy sphinx-astropy
```

Warning

Attempting to use pip to upgrade your installation of `astropy` itself may result in a corrupted installation.

Testing an Installed `astropy`

The easiest way to test if your installed version of `astropy` is running correctly is to use the astropy.test() function:

```
import astropy
astropy.test()
```

The tests should run and print out any failures, which you can report at the Astropy issue tracker.

This way of running the tests may not work if you do it in the `astropy` source distribution. See Testing a Source Code Build of astropy for how to run the tests from the source code directory, or Running Tests for more details.

Building from Source

Prerequisites

You will need a compiler suite and the development headers for Python in order to build `astropy`. You do not need to install any other specific build dependencies (such as Cython or jinja2) since these are declared in the `pyproject.toml` file and will be automatically installed into a temporary build environment by pip.

Prerequisites for Linux

On Linux, using the package manager for your distribution will usually be the easiest route to making sure you have the prerequisites to build `astropy`. In order to build from source, you will need the Python development package for your Linux distribution, as well as pip.

For Debian/Ubuntu:

```
sudo apt-get install python3-dev python3-numpy-dev python3-setuptools
cython3 python3-jinja2 python3-pytest-astropy
```

For Fedora/RHEL:

```
sudo yum install python3-devel python3-numpy python3-setuptools
python3-Cython python3-jinja2 python3-pytest-astropy
```

Note

Building the developer version of `astropy` may require newer versions of the above packages than are available in your distribution's repository. If so, you could either try a more up-to-date distribution (such as Debian `testing`), or install more up-to-date versions of the packages using `pip` or `conda` in a virtual environment.

Prerequisites for Mac OS X

On MacOS X you will need the XCode command line tools which can be installed using:

```
xcode-select --install
```

Follow the onscreen instructions to install the command line tools required. Note that you do **not** need to install the full XCode distribution (assuming you are using MacOS X 10.9 or later).

The instructions for building NumPy from source are a good resource for setting up your environment to build Python packages.

Obtaining the Source Packages

Source Packages

The latest stable source package for `astropy` can be downloaded here.

Development Repository

The latest development version of `astropy` can be cloned from GitHub using this command:

```
git clone git://github.com/astropy/astropy.git
```

If you wish to participate in the development of `astropy`, see Developer Documentation. The present document covers only the basics necessary to installing `astropy`.

Building and Installing

To build and install `astropy` (from the root of the source tree):

```
pip install .
```

If you install in this way and you make changes to the code, you will need to re-run the install command for changes to be reflected. Alternatively, you can use:

```
pip install -e .
```

which installs `astropy` in develop/editable mode – this then means that changes in the code are immediately reflected in the installed version.

Troubleshooting

If you get an error mentioning that you do not have the correct permissions to install `astropy` into the default `site-packages` directory, you can try installing with:

```
pip install . --user
```

which will install into a default directory in your home directory.

External C Libraries

The `astropy` source ships with the C source code of a number of libraries. By default, these internal copies are used to build `astropy`. However, if you wish to use the system-wide installation of one of those libraries, you can set environment variables with the pattern `ASTROPY_USE_SYSTEM_???` to `1` when building/installing the package.

For example, to build `astropy` using the system's expat parser library, use:

```
ASTROPY_USE_SYSTEM_EXPAT=1 pip install -e .
```

To build using all of the system libraries, use:

```
ASTROPY_USE_SYSTEM_ALL=1 pip install -e .
```

The C libraries currently bundled with `astropy` include:

- wcslib see `cextern/wcslib/README` for the bundled version. To use the system version, set `ASTROPY_USE_SYSTEM_WCSLIB=1`.
- cfitsio see `cextern/cfitsio/changes.txt` for the bundled version. To use the system version, set `ASTROPY_USE_SYSTEM_CFITSIO=1`.
- expat see `cextern/expat/README` for the bundled version. To use the system version, set `ASTROPY_USE_SYSTEM_EXPAT=1`.

Installing `astropy` into CASA

If you want to be able to use `astropy` inside CASA, the easiest way is to do so from inside CASA.

First, we need to make sure pip is installed. Start up CASA as normal, and then type:

```
CASA <2>: from setuptools.command import easy_install

CASA <3>: easy_install.main(['--user', 'pip'])
```

Now, quit CASA and re-open it, then type the following to install `astropy`:

```
CASA <2>: import subprocess, sys

CASA <3>: subprocess.check_call([sys.executable, '-m', 'pip',
'install', '--user', 'astropy'])
```

Then close CASA again and open it, and you should be able to import `astropy`:

```
CASA <2>: import astropy
```

Any `astropy` affiliated package can be installed the same way (e.g. the spectral-cube or other packages that may be useful for radio astronomy).

Note

The above instructions have not been tested on all systems. We know of a few examples that do work, but that is not a guarantee that this will work on all systems. If you install `astropy` and begin to encounter issues with CASA, please look at the known CASA issues and if you do not encounter your issue there, please post a new one.

Building Documentation

Note

Building the documentation is in general not necessary unless you are writing new documentation or do not have internet access, because the latest (and archive) versions of Astropy's documentation should be available at docs.astropy.org .

Dependencies

Building the documentation requires the `astropy` source code and some additional packages. The easiest way to build the documentation is to use tox as detailed in Building. If you are happy to do this, you can skip the rest of this section.

On the other hand, if you wish to call Sphinx manually to build the documentation, you will need to make sure that a number of dependencies are installed. If you use conda, the easiest way to install the dependencies is with:

```
conda install -c astropy sphinx-astropy
```

Without conda, you install the dependencies by specifying `[docs]` when installing `astropy` with pip:

```
pip install -e '.[docs]'
```

You can alternatively install the sphinx-astropy package with pip:

```
pip install sphinx-astropy
```

In addition to providing configuration common to packages in the Astropy ecosystem, this package also serves as a way to automatically get the main dependencies, including:

- Sphinx - the main package we use to build the documentation
- astropy-sphinx-theme - the default 'bootstrap' theme used by `astropy` and a number of affiliated packages
- sphinx-automodapi - an extension that makes it easy to automatically generate API documentation
- sphinx-gallery - an extension to generate example galleries
- numpydoc - an extension to parse docstrings in NumPyDoc format
- pillow - used in one of the examples
- Graphviz - generate inheritance graphs (available as a conda package or a system install but not in pip)

Note

Both of the `pip` install methods above do not include Graphviz. If you do not install this package separately then the documentation build process will produce a very large number of lengthy warnings (which can obscure bona fide warnings) and also not generate inheritance graphs.

Building

There are two ways to build the Astropy documentation. The easiest way is to execute the following tox command (from the `astropy` source directory):

```
tox -e build_docs
```

If you do this, you do not need to install any of the documentation dependencies as this will be done automatically. The documentation will be built in the `docs/_build/html` directory, and can be read by pointing a web browser to `docs/_build/html/index.html`.

Alternatively, you can do:

```
cd docs
make html
```

And the documentation will be generated in the same location. Note that this uses the installed version of astropy, so if you want to make sure the current repository version is used, you will need to install it with e.g.:

```
pip install -e .[docs]
```

before changing to the `docs` directory.

In the second way, LaTeX documentation can be generated by using the command:

```
make latex
```

The LaTeX file `Astropy.tex` will be created in the `docs/_build/latex` directory, and can be compiled using `pdflatex`.

Reporting Issues/Requesting Features

As mentioned above, building the documentation depends on a number of Sphinx extensions and other packages. Since it is not always possible to know which package is causing issues or would need to have a new feature implemented, you can open an issue in the core astropy package issue tracker. However, if you wish, you can also open issues in the repositories for some of the dependencies:

- For requests/issues related to the appearance of the docs (e.g. related to the CSS), you can open an issue in the astropy-sphinx-theme issue tracker.
- For requests/issues related to the auto-generated API docs which appear to be general issues rather than an issue with a specific docstring, you can use the sphinx-automodapi issue tracker.
- For issues related to the default configuration (e.g which extensions are enabled by default), you can use the sphinx-astropy issue tracker.

Testing a Source Code Build of `astropy`

The easiest way to run the tests in a source checkout of `astropy` is to use tox:

```
tox -e test-alldeps
```

There are also alternative methods of Running Tests if you would like more control over the testing process.

What’s New in Astropy 4.2?

Overview

Astropy 4.2 is a major release that adds new funcionality since the 4.1 release.

In particular, this release includes:

- Planck 2018 is accepted and now the default cosmology
- Time performance improvements
- Removed ERFA module

In addition to these major changes, Astropy v4.2 includes smaller improvements and bug fixes and significant cleanup, which are described in the Full Changelog. By the numbers:

- 183 issues have been closed since v4.1
- 105 pull requests have been merged since v4.1
- 63 distinct people have contributed code

Planck 2018 is accepted and now the default cosmology

The accepted version of the Planck 2018 cosmological parameters has been included and has become the default cosmology. It is identical to the previous, preliminary version (Planck18_arXiv_v2), which is deprecated and will be removed in a future release.

```
>>> from astropy.cosmology import Planck18
>>> Planck18.age(0)
<Quantity 13.7868853 Gyr>
```

Time performance improvements

The performance for creating a `Time` object from a large array of fixed-format string times was dramatically improved. For ISO and ISOT formats the speed-up is a factor of 25 and for the year day-of-year format the speedup is a factor of 50. This is done with a new C-based time string parser which can also be used for custom user-defined Time formats. For details see Fast C-based Date String Parser.

In addition the performance for creating a scalar `Time` object in a epoch format like `unix`, `unix_tai`, or `cxcsec` was improved by a factor of 4.

Removed ERFA module

The private `_erfa` module has been converted to its own package PyeERFA. It is now a required dependency of astropy, and can be directly imported with `import erfa`. Importing `_erfa` from `astropy` now issues a deprecation warning and will be removed in the future.

Full change log

To see a detailed list of all changes in version v4.2, including changes in API, please see the Full Changelog.

Contributors to the v4.2 release

The people who have contributed to the code for this release are:

- Adrian Price-Whelan
- Albert Y. Shih
- Alex Conley
- Aniket Sanghi *
- Anne Archibald
- Bastian Beischer *
- Ben Greiner *
- Benjamin Winkel
- Bojan Nikolic *
- Brian Soto *
- Brigitta Sipőcz
- Bruce Merry *
- Chris Simpson
- Chun Ly *
- Clara Brasseur
- Clare Shanahan
- David Stansby
- Derek Homeier
- Diego Alonso *
- Ed Slavich
- E. Madison Bray
- Erik Tollerud
- Erin Allard
- Even Rouault *
- Gabriel Perren
- George Galvin *
- Gregory Simonian *
- Hannes Breytenbach
- Hans Moritz Günther
- Inada Naoki *
- James Turner
- Jero Bado *
- Juan Luis Cano Rodríguez
- Kris Stern
- Larry Bradley
- Lauren Glattly
- Lee Spitler *
- Ludwig Schwardt *
- Marten van Kerkwijk
- Matthew Craig
- Maximilian Nöthe
- Michele Costa
- Miguel de Val-Borro
- Mihai Cara
- Nadia Dencheva
- Nathaniel Starkman *
- Nicolas Tessore *
- Nikita Saxena *
- Ole Streicher
- Paul Huwe *
- Peter Yoachim *
- Pey Lian Lim
- Ricardo Fonseca *
- Rui Xue
- Shreyas Bapat
- Simon Conseil
- Stuart Littlefair
- Stuart Mumford
- Thomas Robitaille
- Tom Aldcroft
- Tom Donaldson
- Victoria Dye *
- Zac Hatfield-Dodds *

Importing `astropy` and Sub-packages

In order to encourage consistency among users in importing and using Astropy functionality, we have put together the following guidelines.

Since most of the functionality in Astropy resides in sub-packages, importing `astropy` as:

```
>>> import astropy
```

is not very useful. Instead, it’s best to import the desired sub-package with the syntax:

```
>>> from astropy import subpackage
```

For example, to access the FITS-related functionality, you can import `astropy.io.fits` with:

```
>>> from astropy.io import fits
>>> hdulist = fits.open('data.fits')
```

In specific cases, we have recommended shortcuts in the documentation for specific sub-packages. For example:

```
>>> from astropy import units as u
>>> from astropy import coordinates as coord
>>> coord.SkyCoord(ra=10.68458*u.deg, dec=41.26917*u.deg,
frame='icrs')
<SkyCoord (ICRS): (ra, dec) in deg
    ( 10.68458,  41.26917)>
```

Finally, in some cases, most of the required functionality is contained in a single class (or a few classes). In those cases, the class can be directly imported:

```
>>> from astropy.cosmology import WMAP7
>>> from astropy.table import Table
>>> from astropy.wcs import WCS
```

Note that for clarity, and to avoid any issues, we recommend **never** importing any Astropy functionality using `*`, for example:

```
>>> from astropy.io.fits import *  # NOT recommended
```

Some components of Astropy started off as standalone packages (e.g. PyFITS, PyWCS), so in cases where Astropy needs to be used as a drop-in replacement, the following syntax is also acceptable:

```
>>> from astropy.io import fits as pyfits
```

Getting Started with Sub-packages

Because different sub-packages have very different functionalities, each sub-package has its own getting started guide. These can be found by browsing the sections listed in the User Documentation.

You can also look at docstrings for a particular package or object, or access their documentation using the `find_api_page` function. For example,

```
>>> from astropy import find_api_page
>>> from astropy.units import Quantity
>>> find_api_page(Quantity)
```

will bring up the documentation for the `Quantity` class in your browser.

Example gallery

This gallery of examples shows a variety of relatively small snippets or examples of tasks that can be done with the Astropy core package. Contributions from the community are encouraged!

Longer-form tutorials (or tutorials for affiliated packages) belong at https://learn.astropy.org (and can be submitted at the associated github repository).

astropy.coordinates

General examples of the `astropy.coordinates` subpackage.

Convert a radial velocity to the Galactic Standard of Rest (GSR)

Note

Click here to download the full example code

Convert a radial velocity to the Galactic Standard of Rest (GSR)

Radial or line-of-sight velocities of sources are often reported in a Heliocentric or Solar-system barycentric reference frame. A common transformation incorporates the projection of the Sun's motion along the line-of-sight to the target, hence transforming it to a Galactic rest frame instead (sometimes referred to as the Galactic Standard of Rest, GSR). This transformation depends on the assumptions about the orientation of the Galactic frame relative to the bary- or Heliocentric frame. It also depends on the assumed solar velocity vector. Here we'll demonstrate how to perform this transformation using a sky position and barycentric radial-velocity.

By: Adrian Price-Whelan

License: BSD

Make print work the same in all versions of Python and import the required Astropy packages:

```
import astropy.units as u
import astropy.coordinates as coord
```

Use the latest convention for the Galactocentric coordinates

```
coord.galactocentric_frame_defaults.set('latest')
```

Out:

```
<ScienceState galactocentric_frame_defaults: {'galcen_coord': <ICRS
Coordinate: (ra, dec) in deg...>
```

For this example, let's work with the coordinates and barycentric radial velocity of the star HD 155967, as obtained from Simbad:

```
icrs = coord.SkyCoord(ra=258.58356362*u.deg, dec=14.55255619*u.deg,
                      radial_velocity=-16.1*u.km/u.s, frame='icrs')
```

We next need to decide on the velocity of the Sun in the assumed GSR frame. We'll use the same velocity vector as used in the **`Galactocentric`** frame, and convert it to a **`CartesianRepresentation`** object using the `.to_cartesian()` method of the **`CartesianDifferential`** object `galcen_v_sun`:

```
v_sun = coord.Galactocentric().galcen_v_sun.to_cartesian()
```

We now need to get a unit vector in the assumed Galactic frame from the sky position in the ICRS frame above. We'll use this unit vector to project the solar velocity onto the line-of-sight:

```
gal = icrs.transform_to(coord.Galactic)
cart_data = gal.data.to_cartesian()
unit_vector = cart_data / cart_data.norm()
```

Now we project the solar velocity using this unit vector:

```
v_proj = v_sun.dot(unit_vector)
```

Finally, we add the projection of the solar velocity to the radial velocity to get a GSR radial velocity:

```
rv_gsr = icrs.radial_velocity + v_proj
print(rv_gsr)
```

Out:

```
123.3046008737976 km / s
```

We could wrap this in a function so we can control the solar velocity and re-use the above code:

```
def rv_to_gsr(c, v_sun=None):
    """Transform a barycentric radial velocity to the Galactic
Standard of Rest
    (GSR).

    The input radial velocity must be passed in as a

    Parameters
    ----------
    c : `~astropy.coordinates.BaseCoordinateFrame` subclass instance
        The radial velocity, associated with a sky coordinates, to be
        transformed.
    v_sun : `~astropy.units.Quantity`, optional
        The 3D velocity of the solar system barycenter in the GSR
frame.
        Defaults to the same solar motion as in the
        `~astropy.coordinates.Galactocentric` frame.

    Returns
    -------
    v_gsr : `~astropy.units.Quantity`
        The input radial velocity transformed to a GSR frame.
    """
    if v_sun is None:
        v_sun = coord.Galactocentric().galcen_v_sun.to_cartesian()

    gal = c.transform_to(coord.Galactic)
    cart_data = gal.data.to_cartesian()
    unit_vector = cart_data / cart_data.norm()

    v_proj = v_sun.dot(unit_vector)

    return c.radial_velocity + v_proj

rv_gsr = rv_to_gsr(icrs)
print(rv_gsr)
```

Out:

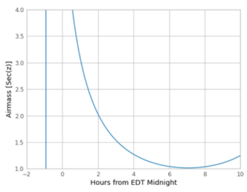
```
123.3046008737976 km / s
```

Total running time of the script: (0 minutes 0.012 seconds)

Download Python source code: rv-to-gsr.py

Download Jupyter notebook: rv-to-gsr.ipynb

Gallery generated by Sphinx-Gallery



Determining and plotting the altitude/azimuth of a celestial object

Note

Click here to download the full example code

Determining and plotting the altitude/azimuth of a celestial object

This example demonstrates coordinate transformations and the creation of visibility curves to assist with observing run planning.

In this example, we make a **`SkyCoord`** instance for M33. The altitude-azimuth coordinates are then found using **`astropy.coordinates.EarthLocation`** and **`astropy.time.Time`** objects.

This example is meant to demonstrate the capabilities of the **`astropy.coordinates`** package. For more convenient and/or complex observation planning, consider the astroplan package.

By: Erik Tollerud, Kelle Cruz

License: BSD

Let’s suppose you are planning to visit picturesque Bear Mountain State Park in New York, USA. You’re bringing your telescope with you (of course), and someone told you M33 is a great target to observe there. You happen to know you’re free at 11:00 pm local time, and you want to know if it will be up. Astropy can answer that.

Import numpy and matplotlib. For the latter, use a nicer set of plot parameters and set up support for plotting/converting quantities.

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.visualization import astropy_mpl_style, quantity_support
plt.style.use(astropy_mpl_style)
quantity_support()
```

Out:

```
<astropy.visualization.units.quantity_support.
<locals>.MplQuantityConverter object at 0x7fb4c6698640>
```

Import the packages necessary for finding coordinates and making coordinate transformations

```
import astropy.units as u
from astropy.time import Time
from astropy.coordinates import SkyCoord, EarthLocation, AltAz
```

`astropy.coordinates.SkyCoord.from_name` uses Simbad to resolve object names and retrieve coordinates.

Get the coordinates of M33:

```
m33 = SkyCoord.from_name('M33')
```

Use `astropy.coordinates.EarthLocation` to provide the location of Bear Mountain and set the time to 11pm EDT on 2012 July 12:

```
bear_mountain = EarthLocation(lat=41.3*u.deg, lon=-74*u.deg,
height=390*u.m)
utcoffset = -4*u.hour  # Eastern Daylight Time
time = Time('2012-7-12 23:00:00') - utcoffset
```

`astropy.coordinates.EarthLocation.get_site_names` and `get_site_names` can be used to get locations of major observatories.

Use `astropy.coordinates` to find the Alt, Az coordinates of M33 at as observed from Bear Mountain at 11pm on 2012 July 12.

```
m33altaz =
m33.transform_to(AltAz(obstime=time,location=bear_mountain))
print("M33's Altitude = {0.alt:.2}".format(m33altaz))
```

Out:

```
M33's Altitude = 0.13 deg
```

This is helpful since it turns out M33 is barely above the horizon at this time. It’s more informative to find M33’s airmass over the course of the night.

Find the alt,az coordinates of M33 at 100 times evenly spaced between 10pm and 7am EDT:

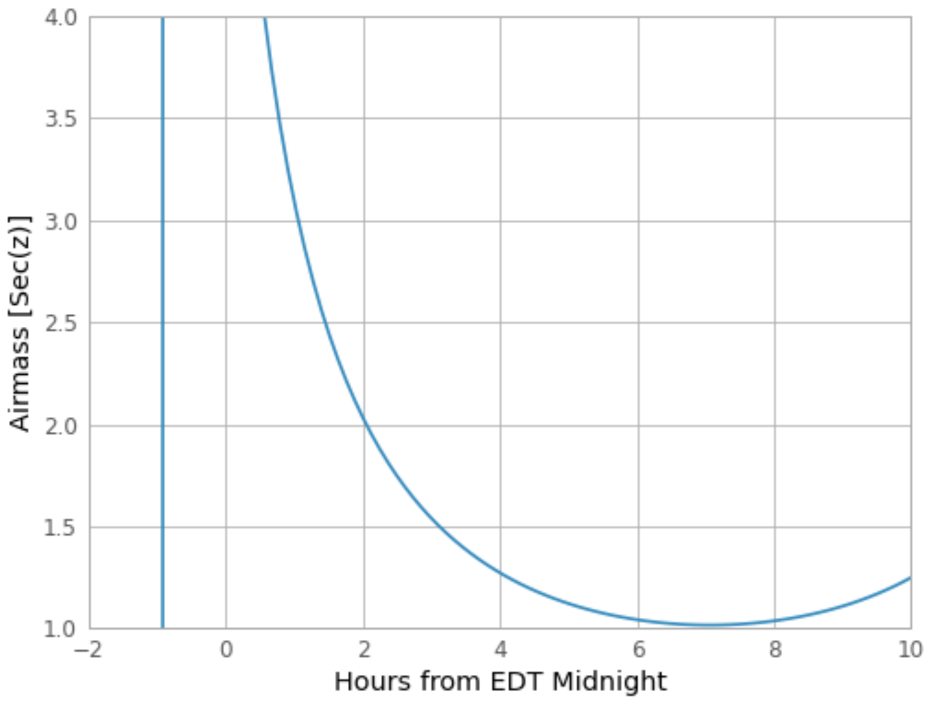
```
midnight = Time('2012-7-13 00:00:00') - utcoffset
delta_midnight = np.linspace(-2, 10, 100)*u.hour
frame_July13night = AltAz(obstime=midnight+delta_midnight,
                          location=bear_mountain)
m33altazs_July13night = m33.transform_to(frame_July13night)
```

convert alt, az to airmass with `secz` attribute:

```
m33airmasss_July13night = m33altazs_July13night.secz
```

Plot the airmass as a function of time:

```
plt.plot(delta_midnight, m33airmasss_July13night)
plt.xlim(-2, 10)
plt.ylim(1, 4)
plt.xlabel('Hours from EDT Midnight')
plt.ylabel('Airmass [Sec(z)]')
plt.show()
```



Use **get_sun** to find the location of the Sun at 1000 evenly spaced times between noon on July 12 and noon on July 13:

```
from astropy.coordinates import get_sun
delta_midnight = np.linspace(-12, 12, 1000)*u.hour
times_July12_to_13 = midnight + delta_midnight
frame_July12_to_13 = AltAz(obstime=times_July12_to_13,
location=bear_mountain)
sunaltazs_July12_to_13 =
get_sun(times_July12_to_13).transform_to(frame_July12_to_13)
```

Do the same with **get_moon** to find when the moon is up. Be aware that this will need to download a 10MB file from the internet to get a precise location of the moon.

```
from astropy.coordinates import get_moon
```

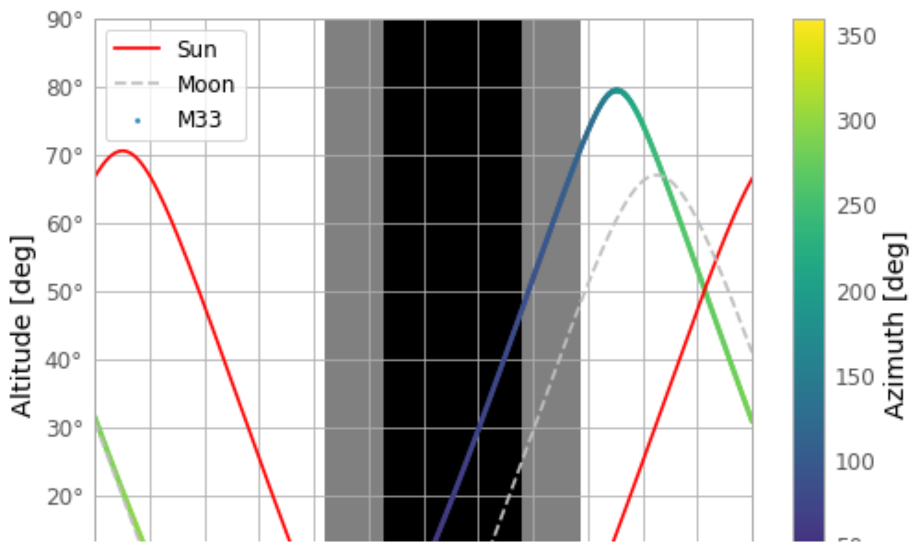
```
moon_July12_to_13 = get_moon(times_July12_to_13)
moonaltazs_July12_to_13 =
moon_July12_to_13.transform_to(frame_July12_to_13)
```

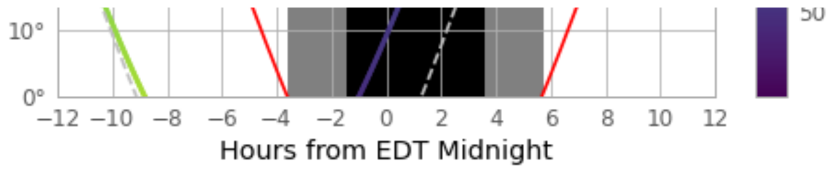
Find the alt,az coordinates of M33 at those same times:

```
m33altazs_July12_to_13 = m33.transform_to(frame_July12_to_13)
```

Make a beautiful figure illustrating nighttime and the altitudes of M33 and the Sun over that time:

```
plt.plot(delta_midnight, sunaltazs_July12_to_13.alt, color='r',
label='Sun')
plt.plot(delta_midnight, moonaltazs_July12_to_13.alt, color=[0.75]*3,
ls='--', label='Moon')
plt.scatter(delta_midnight, m33altazs_July12_to_13.alt,
            c=m33altazs_July12_to_13.az, label='M33', lw=0, s=8,
            cmap='viridis')
plt.fill_between(delta_midnight, 0*u.deg, 90*u.deg,
                 sunaltazs_July12_to_13.alt < -0*u.deg, color='0.5',
zorder=0)
plt.fill_between(delta_midnight, 0*u.deg, 90*u.deg,
                 sunaltazs_July12_to_13.alt < -18*u.deg, color='k',
zorder=0)
plt.colorbar().set_label('Azimuth [deg]')
plt.legend(loc='upper left')
plt.xlim(-12*u.hour, 12*u.hour)
plt.xticks((np.arange(13)*2-12)*u.hour)
plt.ylim(0*u.deg, 90*u.deg)
plt.xlabel('Hours from EDT Midnight')
plt.ylabel('Altitude [deg]')
plt.show()
```



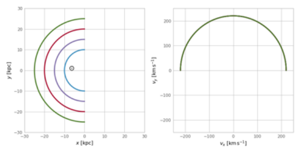


Total running time of the script: (0 minutes 2.956 seconds)

```
Download Python source code: plot_obs-planning.py
```

```
Download Jupyter notebook: plot_obs-planning.ipynb
```

Gallery generated by Sphinx-Gallery



Transforming positions and velocities to and from a Galactocentric frame

Note

Click here to download the full example code

Transforming positions and velocities to and from a Galactocentric frame

This document shows a few examples of how to use and customize the **`Galactocentric`** frame to transform Heliocentric sky positions, distance, proper motions, and radial velocities to a Galactocentric, Cartesian frame, and the same in reverse.

The main configurable parameters of the **`Galactocentric`** frame control the position and velocity of the solar system barycenter within the Galaxy. These are specified by setting the ICRS coordinates of the Galactic center, the distance to the Galactic center (the sun-galactic center line is always assumed to be the x-axis of the Galactocentric frame), and the Cartesian 3-velocity of the sun in the Galactocentric frame. We'll first demonstrate how to customize these values, then show how to set the solar motion instead by inputting the proper motion of Sgr A*.

Note that, for brevity, we may refer to the solar system barycenter as just "the sun" in the examples below.

By: Adrian Price-Whelan

License: BSD

Make **`print`** work the same in all versions of Python, set up numpy, matplotlib, and use a nicer set of plot parameters:

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.visualization import astropy_mpl_style
plt.style.use(astropy_mpl_style)
```

Import the necessary astropy subpackages

```
import astropy.coordinates as coord
import astropy.units as u
```

Let's first define a barycentric coordinate and velocity in the ICRS frame. We'll use the data for the star HD 39881 from the Simbad database:

```
c1 = coord.SkyCoord(ra=89.014303*u.degree, dec=13.924912*u.degree,
                    distance=(37.59*u.mas).to(u.pc, u.parallax()),
                    pm_ra_cosdec=372.72*u.mas/u.yr,
                    pm_dec=-483.69*u.mas/u.yr,
                    radial_velocity=0.37*u.km/u.s,
                    frame='icrs')
```

This is a high proper-motion star; suppose we'd like to transform its position and velocity to a Galactocentric frame to see if it has a large 3D velocity as well. To use the Astropy default solar position and motion parameters, we can simply do:

```
gc1 = c1.transform_to(coord.Galactocentric)
```

From here, we can access the components of the resulting **`Galactocentric`** instance to see the 3D Cartesian velocity components:

```
print(gc1.v_x, gc1.v_y, gc1.v_z)
```

Out:

```
30.254684717897074 km / s 171.29916086104885 km / s 18.19390627095307
km / s
```

The default parameters for the **`Galactocentric`** frame are detailed in the linked documentation, but we can modify the most commonly changes values using the keywords `galcen_distance`, `galcen_v_sun`, and `z_sun` which set the sun-Galactic center distance, the 3D velocity vector of the sun, and the height of the sun above the Galactic midplane, respectively. The velocity of the sun can be specified as an **`Quantity`** object with velocity units and is interepreted as a Cartesian velocity, as in the example below. Note that, as with the positions, the Galactocentric frame is a right-handed system (i.e., the Sun is at negative x values) so `v_x` is opposite of the Galactocentric radial velocity:

```
v_sun = [11.1, 244, 7.25] * (u.km / u.s)  # [vx, vy, vz]
gc_frame = coord.Galactocentric(
    galcen_distance=8*u.kpc,
    galcen_v_sun=v_sun,
    z_sun=0*u.pc)
```

We can then transform to this frame instead, with our custom parameters:

```
gc2 = c1.transform_to(gc_frame)
print(gc2.v_x, gc2.v_y, gc2.v_z)
```

Out:

```
28.427958360720748 km / s 169.69916086104888 km / s 17.70831652451455
km / s
```

It’s sometimes useful to specify the solar motion using the proper motion of Sgr A* instead of Cartesian velocity components. With an assumed distance, we can convert proper motion components to Cartesian velocity components using **`astropy.units`**:

```
galcen_distance = 8*u.kpc
pm_gal_sgrA = [-6.379, -0.202] * u.mas/u.yr # from Reid & Brunthaler
2004
vy, vz = -(galcen_distance * pm_gal_sgrA).to(u.km/u.s,
u.dimensionless_angles())
```

We still have to assume a line-of-sight velocity for the Galactic center, which we will again take to be 11 km/s:

```
vx = 11.1 * u.km/u.s
```

```
v_sun2 = u.Quantity([vx, vy, vz])  # List of Quantity -> a single
Quantity

gc_frame2 = coord.Galactocentric(galcen_distance=galcen_distance,
                                 galcen_v_sun=v_sun2,
                                 z_sun=0*u.pc)
gc3 = c1.transform_to(gc_frame2)
print(gc3.v_x, gc3.v_y, gc3.v_z)
```

Out:

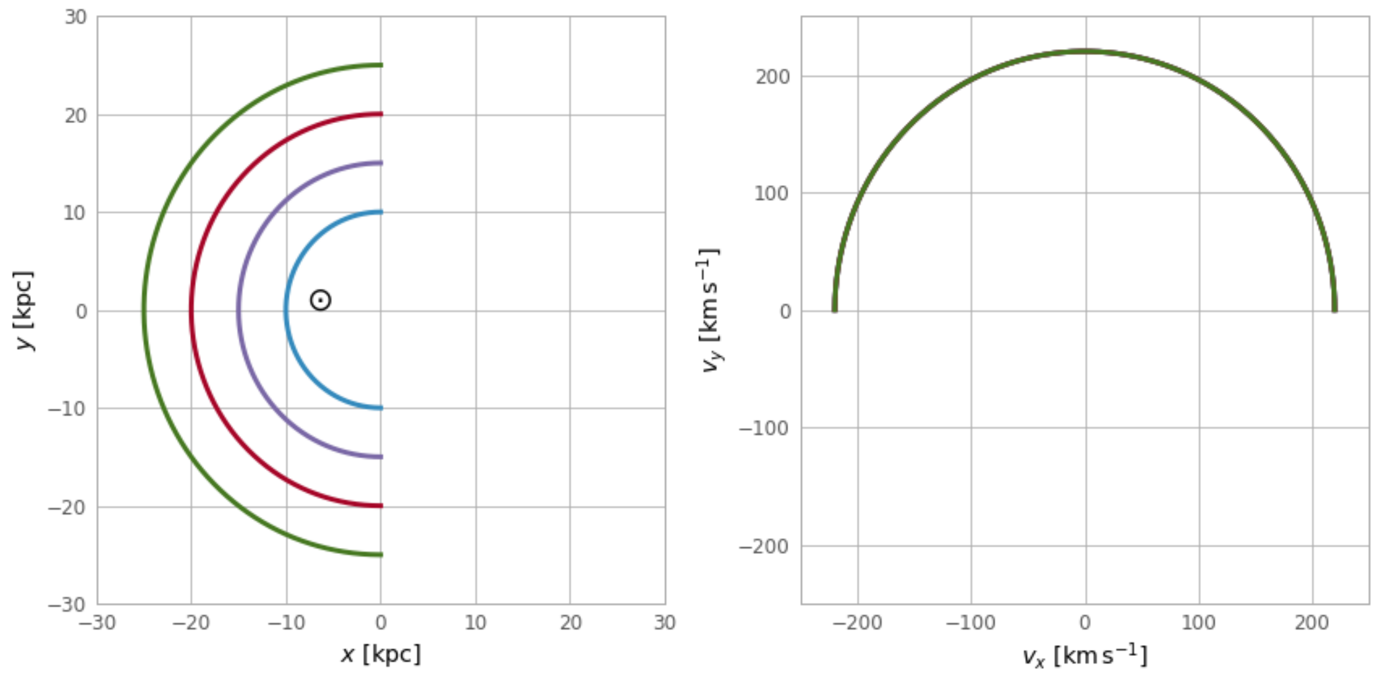
```
28.427958360720748 km / s 167.61484955608267 km / s 18.118916793584443
km / s
```

The transformations also work in the opposite direction. This can be useful for transforming simulated or theoretical data to observable quantities. As an example, we'll generate 4 theoretical circular orbits at different Galactocentric radii with the same circular velocity, and transform them to Heliocentric coordinates:

```
ring_distances = np.arange(10, 25+1, 5) * u.kpc
circ_velocity = 220 * u.km/u.s

phi_grid = np.linspace(90, 270, 512) * u.degree # grid of azimuths
ring_rep = coord.CylindricalRepresentation(
    rho=ring_distances[:,np.newaxis],
    phi=phi_grid[np.newaxis],
    z=np.zeros_like(ring_distances)[:,np.newaxis])

angular_velocity = (-circ_velocity / ring_distances).to(u.mas/u.yr,

u.dimensionless_angles())
ring_dif = coord.CylindricalDifferential(
    d_rho=np.zeros(phi_grid.shape)[np.newaxis]*u.km/u.s,
    d_phi=angular_velocity[:,np.newaxis],
    d_z=np.zeros(phi_grid.shape)[np.newaxis]*u.km/u.s
)

ring_rep = ring_rep.with_differentials(ring_dif)
gc_rings = coord.SkyCoord(ring_rep, frame=coord.Galactocentric)
```

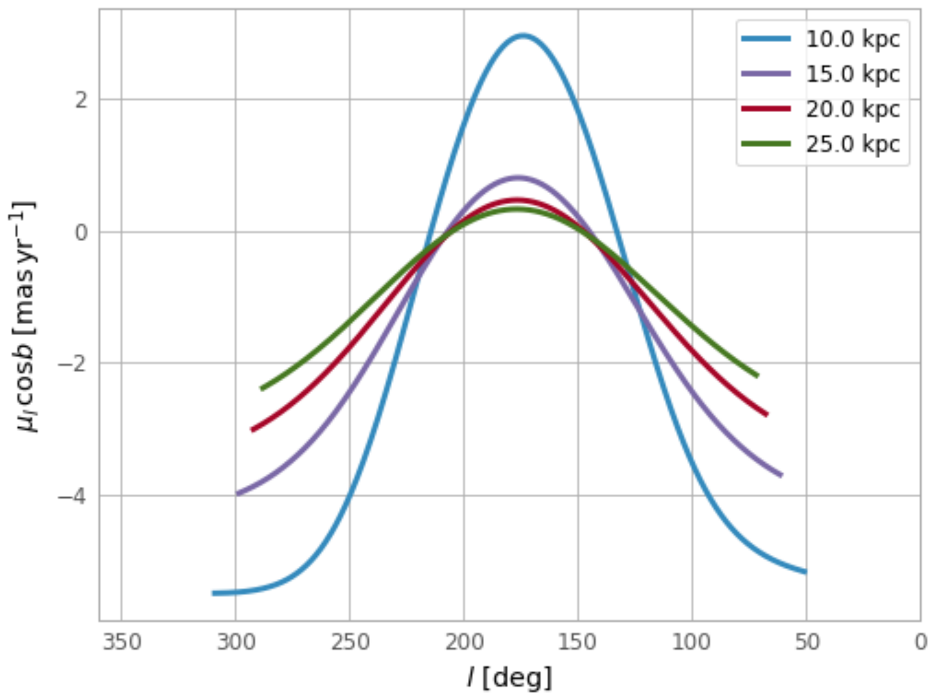
First, let's visualize the geometry in Galactocentric coordinates. Here are the positions and velocities of the rings; note that in the velocity plot, the velocities of the 4 rings are identical and thus overlaid under the same curve:

```
fig,axes = plt.subplots(1, 2, figsize=(12,6))

# Positions
axes[0].plot(gc_rings.x.T, gc_rings.y.T, marker='None', linewidth=3)
axes[0].text(-8., 0, r'$\odot$', fontsize=20)

axes[0].set_xlim(-30, 30)
axes[0].set_ylim(-30, 30)

axes[0].set_xlabel('$x$ [kpc]')
axes[0].set_ylabel('$y$ [kpc]')

# Velocities
axes[1].plot(gc_rings.v_x.T, gc_rings.v_y.T, marker='None',
linewidth=3)

axes[1].set_xlim(-250, 250)
axes[1].set_ylim(-250, 250)

axes[1].set_xlabel('$v_x$
[{0}]'.format((u.km/u.s).to_string("latex_inline")))
axes[1].set_ylabel('$v_y$
[{0}]'.format((u.km/u.s).to_string("latex_inline")))

fig.tight_layout()

plt.show()
```



Now we can transform to Galactic coordinates and visualize the rings in observable coordinates:

```
gal_rings = gc_rings.transform_to(coord.Galactic)

fig,ax = plt.subplots(1, 1, figsize=(8,6))
for i in range(len(ring_distances)):
    ax.plot(gal_rings[i].l.degree, gal_rings[i].pm_l_cosb.value,
            label=str(ring_distances[i]), marker='None', linewidth=3)

ax.set_xlim(360, 0)

ax.set_xlabel('$l$ [deg]')
ax.set_ylabel(r'$\mu_l \, \cos b$
[{0}]'.format((u.mas/u.yr).to_string('latex_inline')))

ax.legend()

plt.show()
```



Total running time of the script: (0 minutes 0.523 seconds)

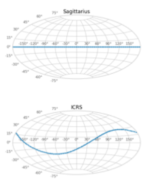
Download Python source code: plot_galactocentric-frame.py

Download Jupyter notebook: plot_galactocentric-frame.ipynb

Gallery generated by Sphinx-Gallery

Note

Click here to download the full example code



Create a new coordinate class (for the Sagittarius stream)

Create a new coordinate class (for the Sagittarius stream)

This document describes in detail how to subclass and define a custom spherical coordinate frame, as discussed in Defining a New Frame and the docstring for `BaseCoordinateFrame`. In this example, we will define a coordinate system defined by the plane of orbit of the Sagittarius Dwarf Galaxy (hereafter Sgr; as defined in Majewski et al. 2003). The Sgr coordinate system is often referred to in terms of two angular coordinates, \(\Lambda,B\).

To do this, we need to define a subclass of `BaseCoordinateFrame` that knows the names and units of the coordinate system angles in each of the supported representations. In this case we support `SphericalRepresentation` with “Lambda” and “Beta”. Then we have to define the transformation from this coordinate system to some other built-in system. Here we will use Galactic coordinates, represented by the `Galactic` class.

See Also

- The gala package, which defines a number of Astropy coordinate frames for stellar stream coordinate systems.
- Majewski et al. 2003, “A Two Micron All Sky Survey View of the Sagittarius Dwarf Galaxy. I. Morphology of the Sagittarius Core and Tidal Arms”, https://arxiv.org/abs/astro-ph/0304198
- Law & Majewski 2010, “The Sagittarius Dwarf Galaxy: A Model for Evolution in a Triaxial Milky Way Halo”, https://arxiv.org/abs/1003.1132
- David Law’s Sgr info page https://www.stsci.edu/~dlaw/Sgr/

By: Adrian Price-Whelan, Erik Tollerud

License: BSD

Make `print` work the same in all versions of Python, set up numpy, matplotlib, and use a nicer set of plot parameters:

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.visualization import astropy_mpl_style
plt.style.use(astropy_mpl_style)
```

Import the packages necessary for coordinates

```
from astropy.coordinates import frame_transform_graph
from astropy.coordinates.matrix_utilities import rotation_matrix,
matrix_product, matrix_transpose
import astropy.coordinates as coord
import astropy.units as u
```

The first step is to create a new class, which we'll call `Sagittarius` and make it a subclass of **BaseCoordinateFrame**:

```
class Sagittarius(coord.BaseCoordinateFrame):
    """
    A Heliocentric spherical coordinate system defined by the orbit
    of the Sagittarius dwarf galaxy, as described in
        https://ui.adsabs.harvard.edu/abs/2003ApJ...599.1082M
    and further explained in
        https://www.stsci.edu/~dlaw/Sgr/.

    Parameters
    ----------
    representation : `~astropy.coordinates.BaseRepresentation` or
None
        A representation object or None to have no data (or use the
other keywords)
    Lambda : `~astropy.coordinates.Angle`, optional, must be keyword
        The longitude-like angle corresponding to Sagittarius' orbit.
    Beta : `~astropy.coordinates.Angle`, optional, must be keyword
        The latitude-like angle corresponding to Sagittarius' orbit.
    distance : `Quantity`, optional, must be keyword
        The Distance for this object along the line-of-sight.
    pm_Lambda_cosBeta : :class:`~astropy.units.Quantity`, optional,
must be keyword
        The proper motion along the stream in ``Lambda`` (including
the
        ``cos(Beta)`` factor) for this object (``pm_Beta`` must also
be given).
    pm_Beta : :class:`~astropy.units.Quantity`, optional, must be
keyword
        The proper motion in Declination for this object
(``pm_ra_cosdec`` must
        also be given).
    radial_velocity : :class:`~astropy.units.Quantity`, optional,
must be keyword
        The radial velocity of this object.

    """

    default_representation = coord.SphericalRepresentation
    default_differential = coord.SphericalCosLatDifferential

    frame_specific_representation_info = {
        coord.SphericalRepresentation: [
            coord.RepresentationMapping('lon', 'Lambda'),
            coord.RepresentationMapping('lat', 'Beta'),
            coord.RepresentationMapping('distance', 'distance')]
    }
```

Breaking this down line-by-line, we define the class as a subclass of `BaseCoordinateFrame`. Then we include a descriptive docstring. The final lines are class-level attributes that specify the default representation for the data, default differential for the velocity information, and mappings from the attribute names used by representation objects to the names that are to be used by the `Sagittarius` frame. In this case we override the names in the spherical representations but don't do anything with other representations like cartesian or cylindrical.

Next we have to define the transformation from this coordinate system to some other built-in coordinate system; we will use Galactic coordinates. We can do this by defining functions that return transformation matrices, or by simply defining a function that accepts a coordinate and returns a new coordinate in the new system. Because the transformation to the Sagittarius coordinate system is just a spherical rotation from Galactic coordinates, we'll just define a function that returns this matrix. We'll start by constructing the transformation matrix using pre-determined Euler angles and the `rotation_matrix` helper function:

```
SGR_PHI = (180 + 3.75) * u.degree # Euler angles (from Law & Majewski
2010)
SGR_THETA = (90 - 13.46) * u.degree
SGR_PSI = (180 + 14.111534) * u.degree

# Generate the rotation matrix using the x-convention (see Goldstein)
D = rotation_matrix(SGR_PHI, "z")
C = rotation_matrix(SGR_THETA, "x")
B = rotation_matrix(SGR_PSI, "z")
A = np.diag([1.,1.,-1.])
SGR_MATRIX = matrix_product(A, B, C, D)
```

Since we already constructed the transformation (rotation) matrix above, and the inverse of a rotation matrix is just its transpose, the required transformation functions are very simple:

```
@frame_transform_graph.transform(coord.StaticMatrixTransform,
coord.Galactic, Sagittarius)
def galactic_to_sgr():
    """ Compute the transformation matrix from Galactic spherical to
        heliocentric Sgr coordinates.
    """
    return SGR_MATRIX
```

The decorator `@frame_transform_graph.transform(coord.StaticMatrixTransfor` `coord.Galactic, Sagittarius)` registers this function on the `frame_transform_graph` as a coordinate transformation. Inside the function, we simply return the previously defined rotation matrix.

We then register the inverse transformation by using the transpose of the rotation matrix (which is faster to compute than the inverse):

```
@frame_transform_graph.transform(coord.StaticMatrixTransform,
Sagittarius, coord.Galactic)
def sgr_to_galactic():
    """ Compute the transformation matrix from heliocentric Sgr
coordinates to
        spherical Galactic.
    """
    return matrix_transpose(SGR_MATRIX)
```

Now that we've registered these transformations between `Sagittarius` and **`Galactic`**, we can transform between *any* coordinate system and `Sagittarius` (as long as the other system has a path to transform to **`Galactic`**). For example, to transform from ICRS coordinates to `Sagittarius`, we would do:

```
icrs = coord.SkyCoord(280.161732*u.degree, 11.91934*u.degree,
frame='icrs')
sgr = icrs.transform_to(Sagittarius)
print(sgr)
```

Out:

```
<SkyCoord (Sagittarius): (Lambda, Beta) in deg
    (346.81830652, -39.28360407)>
```

Or, to transform from the `Sagittarius` frame to ICRS coordinates (in this case, a line along the `Sagittarius` x-y plane):

```
sgr = coord.SkyCoord(Lambda=np.linspace(0, 2*np.pi, 128)*u.radian,
                     Beta=np.zeros(128)*u.radian,
frame='sagittarius')
icrs = sgr.transform_to(coord.ICRS)
print(icrs)
```

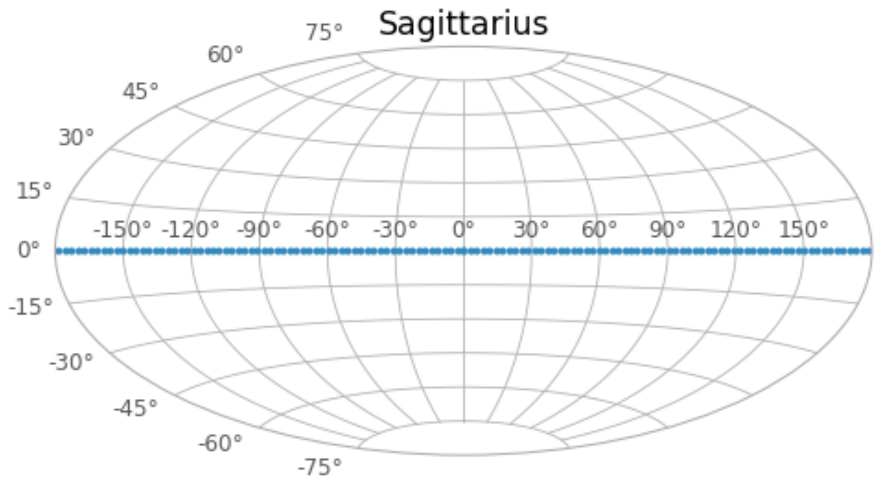
Out:

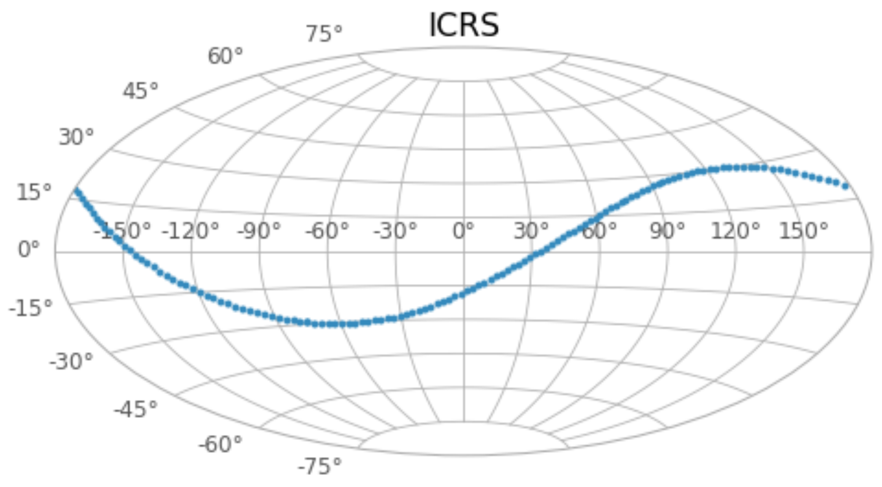
```
<SkyCoord (ICRS): (ra, dec) in deg
    [(284.03876751, -29.00408353), (287.24685769, -29.44848352),
     (290.48068369, -29.81535572), (293.7357366 , -30.1029631 ),
     (297.00711066, -30.30991693), (300.28958688, -30.43520293),
     (303.57772919, -30.47820084), (306.86598944, -30.43869669),
     (310.14881715, -30.31688708), (313.42076929, -30.11337526),
     (316.67661568, -29.82915917), (319.91143548, -29.46561215),
     (323.12070147, -29.02445708), (326.30034928, -28.50773532),
     (329.44683007, -27.9177717 ), (332.55714589, -27.257137  ),
     (335.62886847, -26.52860943), (338.66014233, -25.73513624),
     (341.64967439, -24.87979679), (344.59671212, -23.96576781),
     (347.50101283, -22.99629167), (350.36280652, -21.97464811),
     (353.18275454, -20.90412969), (355.96190618, -19.78802107),
     (358.70165491, -18.62958199), (  1.40369557, -17.43203397),
     (  4.06998374, -16.19855028), (  6.70269788, -14.93224899),
     (  9.30420479, -13.63618882), ( 11.87702861, -12.31336727),
     ( 14.42382347, -10.96672102), ( 16.94734952,  -9.59912794),
     ( 19.45045241,  -8.21341071), ( 21.93604568,  -6.81234162),
     ( 24.40709589,  -5.39864845), ( 26.86661004,  -3.97502106),
     ( 29.31762493,  -2.54411871), ( 31.76319801,  -1.10857781),
     ( 34.20639942,   0.32898001), ( 36.65030466,   1.76593955),
     ( 39.09798768,   3.19968374), ( 41.55251374,   4.6275852 ),
     ( 44.01693189,   6.04699804), ( 46.49426651,   7.45524993),
     ( 48.98750752,   8.84963453), ( 51.4995989 ,  10.22740448),
     ( 54.03342512,  11.58576509), ( 56.59179508,  12.92186896),
     ( 59.17742314,  14.23281165), ( 61.79290712,  15.51562883),
     ( 64.44070278,  16.76729487), ( 67.12309478,  17.98472356),
     ( 69.84216409,  19.16477088), ( 72.59975183,  20.30424045),
     ( 75.39742013,  21.3998918 ), ( 78.23641033,  22.44845192),
     ( 81.11759966,  23.44663022), ( 84.04145735,  24.39113719),
     ( 87.00800203,  25.27870692), ( 90.01676196,  26.10612335),
     ( 93.06674057,  26.87025019), ( 96.15638947,  27.56806406),
     ( 99.28359159,  28.19669038), (102.44565666,  28.75344107),
     (105.63933131,  29.23585315), (108.86082534,  29.64172698),
     (112.105855  ,  29.96916281), (115.36970341,  30.21659414),
     (118.64729687,  30.38281659), (121.93329519,  30.46701088),
     (125.22219273,  30.46875885), (128.50842634,  30.38805179),
     (131.78648572,  30.22529063), (135.05102157,  29.98127794),
     (138.29694697,  29.6572022 ), (141.51952827,  29.2546151 ),
     (144.71446203,  28.77540295), (147.87793614,  28.22175338),
     (151.00667382,  27.59611901), (154.09796066,  26.90117914),
     (157.14965528,  26.13980125), (160.16018547,  25.31500315),
     (163.12853176,  24.42991703), (166.05420084,  23.48775622),
     (168.93719133,  22.49178507), (171.77795423,  21.44529257),
     (174.57735037,  20.35156967), (177.33660656,  19.21389046),
     (180.05727218,  18.03549704), (182.74117737,  16.81958784),
     (185.39039367,  15.56930924), (188.00719783,  14.28774998),
     (190.59403895,  12.97793826), (193.15350938,  11.64284103),
     (195.68831902,  10.28536518), (198.20127316,   8.90836046),
     (200.69525342,   7.51462369), (203.17320154,   6.10690412),
     (205.63810576,   4.6879097 ), (208.09298919,   3.26031403),
     (210.54090002,   1.82676397), (212.984903  ,   0.38988751),
     (215.42807182,  -1.04769799), (217.87348209,  -2.48337744),
     (220.32420429,  -3.91452965), (222.7832966 ,  -5.338519  ),
     (225.25379684,  -6.75268736), (227.73871349,  -8.15434631),
     (230.24101506,  -9.54076983), (232.76361762, -10.90918763),
     (235.30937003, -12.25677927), (237.88103647, -13.58066929),
     (240.48127601, -14.87792359), (243.11261883, -16.14554723),
     (245.777439  , -17.38048408), (248.47792364, -18.57961852),
     (251.2160385 , -19.7397795 ), (253.9934903 , -20.85774736),
     (256.81168612, -21.93026371), (259.67169071, -22.95404466),
     (262.57418275, -23.92579758), (265.51941137, -24.84224172),
     (268.50715471, -25.70013256), (271.53668252, -26.49628998),
     (274.6067251 , -27.22762983), (277.71545113, -27.89119849),
     (280.86045662, -28.48420985), (284.03876751, -29.00408353)]>
```

As an example, we'll now plot the points in both coordinate systems:

```
fig, axes = plt.subplots(2, 1, figsize=(8, 10),
                         subplot_kw={'projection': 'aitoff'})

axes[0].set_title("Sagittarius")
axes[0].plot(sgr.Lambda.wrap_at(180*u.deg).radian, sgr.Beta.radian,
             linestyle='none', marker='.')
```

```
axes[1].set_title("ICRS")
axes[1].plot(icrs.ra.wrap_at(180*u.deg).radian, icrs.dec.radian,
             linestyle='none', marker='.')

plt.show()
```

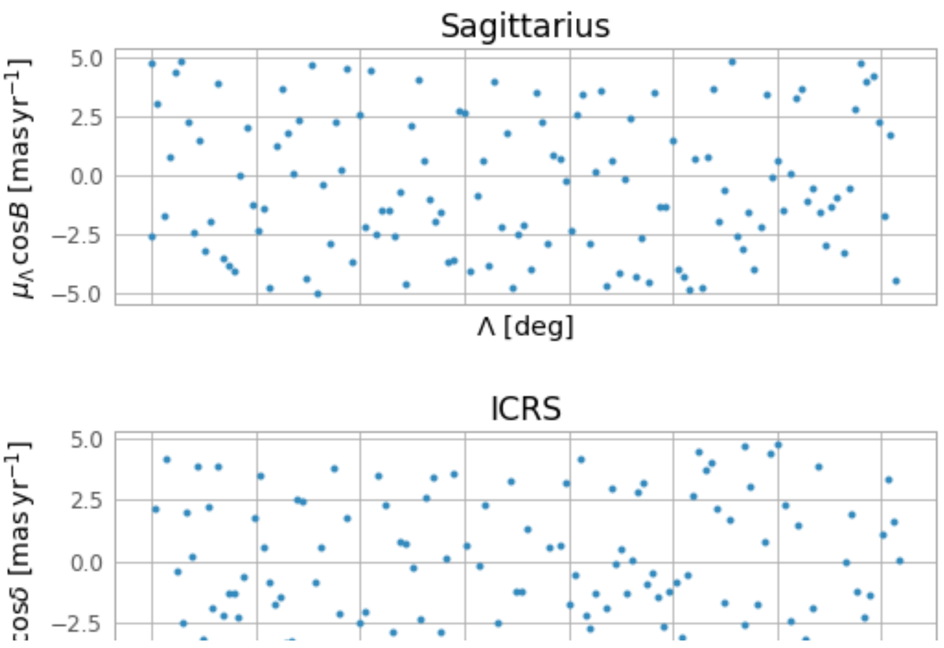


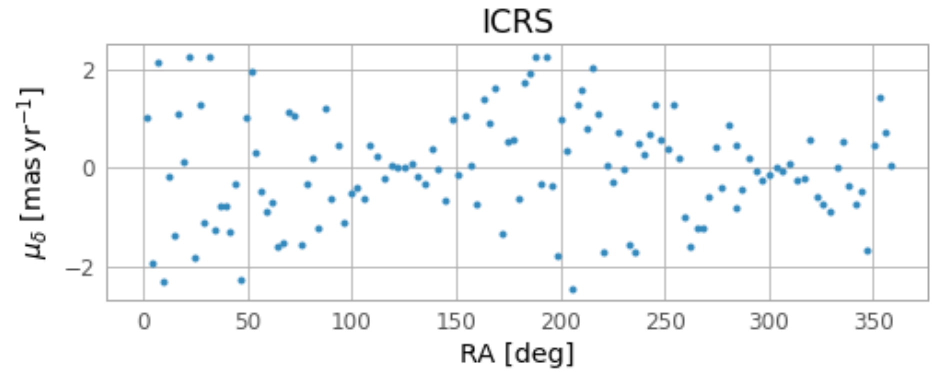


This particular transformation is just a spherical rotation, which is a special case of an Affine transformation with no vector offset. The transformation of velocity components is therefore natively supported as well:

```
sgr = coord.SkyCoord(Lambda=np.linspace(0, 2*np.pi, 128)*u.radian,
                     Beta=np.zeros(128)*u.radian,
                     pm_Lambda_cosBeta=np.random.uniform(-5, 5,
128)*u.mas/u.yr,
                     pm_Beta=np.zeros(128)*u.mas/u.yr,
                     frame='sagittarius')
```

```
icrs = sgr.transform_to(coord.ICRS)
print(icrs)

fig, axes = plt.subplots(3, 1, figsize=(8, 10), sharex=True)

axes[0].set_title("Sagittarius")
axes[0].plot(sgr.Lambda.degree,
             sgr.pm_Lambda_cosBeta.value,
             linestyle='none', marker='.')
axes[0].set_xlabel(r"$\Lambda$ [deg]")
axes[0].set_ylabel(r"$\mu_\Lambda \, \cos B$ [{0}]"

.format(sgr.pm_Lambda_cosBeta.unit.to_string('latex_inline')))

axes[1].set_title("ICRS")
axes[1].plot(icrs.ra.degree, icrs.pm_ra_cosdec.value,
             linestyle='none', marker='.')
axes[1].set_ylabel(r"$\mu_\alpha \, \cos\delta$ [{0}]"

.format(icrs.pm_ra_cosdec.unit.to_string('latex_inline')))

axes[2].set_title("ICRS")
axes[2].plot(icrs.ra.degree, icrs.pm_dec.value,
             linestyle='none', marker='.')
axes[2].set_xlabel("RA [deg]")
axes[2].set_ylabel(r"$\mu_\delta$ [{0}]"

.format(icrs.pm_dec.unit.to_string('latex_inline')))

plt.show()
```





Out:

```
<SkyCoord (ICRS): (ra, dec) in deg
    [(284.03876751, -29.00408353), (287.24685769, -29.44848352),
     (290.48068369, -29.81535572), (293.7357366 , -30.1029631 ),
     (297.00711066, -30.30991693), (300.28958688, -30.43520293),
     (303.57772919, -30.47820084), (306.86598944, -30.43869669),
     (310.14881715, -30.31688708), (313.42076929, -30.11337526),
     (316.67661568, -29.82915917), (319.91143548, -29.46561215),
     (323.12070147, -29.02445708), (326.30034928, -28.50773532),
     (329.44683007, -27.9177717 ), (332.55714589, -27.257137  ),
     (335.62886847, -26.52860943), (338.66014233, -25.73513624),
     (341.64967439, -24.87979679), (344.59671212, -23.96576781),
     (347.50101283, -22.99629167), (350.36280652, -21.97464811),
     (353.18275454, -20.90412969), (355.96190618, -19.78802107),
     (358.70165491, -18.62958199), (  1.40369557, -17.43203397),
     (  4.06998374, -16.19855028), (  6.70269788, -14.93224899),
     (  9.30420479, -13.63618882), ( 11.87702861, -12.31336727),
     ( 14.42382347, -10.96672102), ( 16.94734952,  -9.59912794),
     ( 19.45045241,  -8.21341071), ( 21.93604568,  -6.81234162),
     ( 24.40709589,  -5.39864845), ( 26.86661004,  -3.97502106),
     ( 29.31762493,  -2.54411871), ( 31.76319801,  -1.10857781),
     ( 34.20639942,   0.32898001), ( 36.65030466,   1.76593955),
     ( 39.09798768,   3.19968374), ( 41.55251374,   4.6275852 ),
     ( 44.01693189,   6.04699804), ( 46.49426651,   7.45524993),
     ( 48.98750752,   8.84963453), ( 51.4995989 ,  10.22740448),
     ( 54.03342512,  11.58576509), ( 56.59179508,  12.92186896),
     ( 59.17742314,  14.23281165), ( 61.79290712,  15.51562883),
     ( 64.44070278,  16.76729487), ( 67.12309478,  17.98472356),
     ( 69.84216409,  19.16477088), ( 72.59975183,  20.30424045),
     ( 75.39742013,  21.3998918 ), ( 78.23641033,  22.44845192),
     ( 81.11759966,  23.44663022), ( 84.04145735,  24.39113719),
     ( 87.00800203,  25.27870692), ( 90.01676196,  26.10612335),
     ( 93.06674057,  26.87025019), ( 96.15638947,  27.56806406),
     ( 99.28359159,  28.19669038), (102.44565666,  28.75344107),
     (105.63933131,  29.23585315), (108.86082534,  29.64172698),
     (112.105855  ,  29.96916281), (115.36970341,  30.21659414),
     (118.64729687,  30.38281659), (121.93329519,  30.46701088),
     (125.22219273,  30.46875885), (128.50842634,  30.38805179),
     (131.78648572,  30.22529063), (135.05102157,  29.98127794),
     (138.29694697,  29.6572022 ), (141.51952827,  29.2546151 ),
     (144.71446203,  28.77540295), (147.87793614,  28.22175338),
     (151.00667382,  27.59611901), (154.09796066,  26.90117914),
     (157.14965528,  26.13980125), (160.16018547,  25.31500315),
     (163.12853176,  24.42991703), (166.05420084,  23.48775622),
     (168.93719133,  22.49178507), (171.77795423,  21.44529257),
     (174.57735037,  20.35156967), (177.33660656,  19.21389046),
     (180.05727218,  18.03549704), (182.74117737,  16.81958784),
     (185.39039367,  15.56930924), (188.00719783,  14.28774998),
     (190.59403895,  12.97793826), (193.15350938,  11.64284103),
     (195.68831902,  10.28536518), (198.20127316,   8.90836046),
     (200.69525342,   7.51462369), (203.17320154,   6.10690412),
     (205.63810576,   4.6879097 ), (208.09298919,   3.26031403),
     (210.54090002,   1.82676397), (212.984903  ,   0.38988751),
     (215.42807182,  -1.04769799), (217.87348209,  -2.48337744),
     (220.32420429,  -3.91452965), (222.7832966 ,  -5.338519  ),
     (225.25379684,  -6.75268736), (227.73871349,  -8.15434631),
     (230.24101506,  -9.54076983), (232.76361762, -10.90918763),
     (235.30937003, -12.25677927), (237.88103647, -13.58066929),
     (240.48127601, -14.87792359), (243.11261883, -16.14554723),
     (245.777439  , -17.38048408), (248.47792364, -18.57961852),
     (251.2160385 , -19.7397795 ), (253.9934903 , -20.85774736),
     (256.81168612, -21.93026371), (259.67169071, -22.95404466),
     (262.57418275, -23.92579758), (265.51941137, -24.84224172),
     (268.50715471, -25.70013256), (271.53668252, -26.49628998),
     (274.6067251 , -27.22762983), (277.71545113, -27.89119849),
     (280.86045662, -28.48420985), (284.03876751, -29.00408353)]
 (pm_ra_cosdec, pm_dec) in mas / yr
    [(-2.56302619,  4.42730660e-01), ( 3.03790048, -4.39592780e-01),
     (-1.74159821,  2.02572370e-01), ( 0.77723376, -6.81172865e-02),
     ( 4.39313491, -2.58108976e-01), ( 4.81938745, -1.43237750e-01),
     ( 2.29044968, -1.41268720e-03), (-2.43077916, -6.92507501e-02),
     ( 1.49349425,  8.59136866e-02), (-3.16485168, -2.73508651e-01),
     (-1.91591571, -2.20530705e-01), ( 3.86453743,  5.54587509e-01),
     (-3.48448515, -5.97790864e-01), (-3.78097734, -7.53129434e-01),
     (-3.98821822, -9.02664818e-01), (-0.04129046, -1.04432976e-02),
     ( 1.96729285,  5.48665572e-01), (-1.2321328 , -3.74792796e-01),
     (-2.23396783, -7.34363767e-01), (-1.36206351, -4.80080831e-01),
     (-4.4631792 , -1.67522259e+00), ( 1.13620319,  4.51415160e-01),
     ( 3.38415139,  1.41554897e+00), ( 1.65710432,  7.26226598e-01),
     ( 0.06720867,  3.07238097e-02), ( 2.12504405,  1.00920905e+00),
     (-3.96092514, -1.94687932e+00), ( 4.1964179 ,  2.12728101e+00),
     (-4.42815894, -2.30748597e+00), (-0.36578103, -1.95321390e-01),
     (-2.52942416, -1.37997403e+00), ( 1.98230493,  1.10179944e+00),
     ( 0.2231882 ,  1.26035724e-01), ( 3.91926885,  2.24265580e+00),
     (-3.15680867, -1.82564328e+00), ( 2.22244799,  1.29571050e+00),
     (-1.92474051, -1.12842723e+00), ( 3.86338205,  2.27207345e+00),
     (-2.1708685 , -1.27754607e+00), (-1.29985031, -7.63591635e-01),
     (-1.30212561, -7.61689701e-01), (-2.23763636, -1.30015640e+00),
     (-0.60088965, -3.45931900e-01), (-3.99998665, -2.27579318e+00),
     ( 1.81273802,  1.01659893e+00), ( 3.53944416,  1.95127594e+00),
     ( 0.55048634,  2.97500038e-01), (-0.87420013, -4.61792351e-01),
     (-1.73664295, -8.93966776e-01), (-1.4302138 , -7.15148435e-01),
     (-3.32820364, -1.61107441e+00), (-3.28037544, -1.53167579e+00),
     ( 2.52409705,  1.13239038e+00), ( 2.46271137,  1.05710649e+00),
     (-3.79881871, -1.55298618e+00), (-0.80592427, -3.12190815e-01),
     ( 0.5524415 ,  2.01632428e-01), (-3.6249229 , -1.23865548e+00),
     ( 3.79231866,  1.20440340e+00), (-2.11292557, -6.18448994e-01),
     ( 1.74851415,  4.67009574e-01), (-4.64232787, -1.11799886e+00),
     (-2.47219919, -5.28990510e-01), (-2.07616462, -3.87413404e-01),
     (-3.92449393, -6.23086884e-01), ( 3.51187567,  4.58469701e-01),
     ( 2.2817848 ,  2.32761619e-01), (-2.86521217, -2.09788357e-01),
     ( 0.84509531,  3.73959021e-02), ( 0.70851238,  1.07506869e-02),
     (-0.25795084,  3.59594739e-03), (-2.33363336,  1.00393756e-01),
     ( 2.60786628, -1.87753366e-01), ( 3.45647655, -3.48390293e-01),
     (-2.90359286,  3.75566045e-01), ( 0.12721268, -2.00462462e-02),
     ( 3.56951368, -6.61895682e-01), (-4.578107  ,  9.74340590e-01),
     ( 0.63801943, -1.52933981e-01), (-4.00313886,  1.06479330e+00),
     (-0.18864275,  5.50133467e-02), ( 2.32284125, -7.35297768e-01),
     (-4.10435233,  1.39835386e+00), (-2.52998779,  9.20955339e-01),
     (-4.22859302,  1.63409915e+00), ( 3.25826716, -1.32911032e+00),
     (-1.20177126,  5.14838948e-01), (-1.2512829 ,  5.60365133e-01),
     ( 1.3418169 , -6.25513356e-01), (-3.61334494,  1.74656301e+00),
     (-3.8471524 ,  1.92118087e+00), (-4.35542158,  2.23941883e+00),
     ( 0.6155346 , -3.24817956e-01), (-4.21344578,  2.27501490e+00),
     ( 0.64906697, -3.57545954e-01), ( 3.18151583, -1.78302691e+00),
     (-1.72472915,  9.80738575e-01), (-0.56659908,  3.26044956e-01),
     ( 4.21476306, -2.44811387e+00), (-2.20684615,  1.29061144e+00),
     (-2.68517388,  1.57718948e+00), (-1.32232068,  7.78159429e-01),
     (-3.44779794,  2.02782422e+00), (-1.87140064,  1.09735501e+00),
     ( 2.9674062 , -1.73052543e+00), (-0.06185675,  3.57870244e-02),
     ( 0.51674235, -2.95835008e-01), (-1.28396124,  7.25505165e-01),
     ( 0.0863801 , -4.80464823e-02), ( 2.86144289, -1.56243343e+00),
     ( 3.22915614, -1.72599027e+00), (-0.95305005,  4.97173782e-01),
     (-0.50381019,  2.55710567e-01), (-1.4143179 ,  6.96125478e-01),
     (-2.67553466,  1.27259020e+00), (-1.2128097 ,  5.55365538e-01),
     (-0.84816125,  3.72404497e-01), (-3.07188277,  1.28759366e+00),
     (-0.50683459,  2.01826916e-01), ( 2.66344179, -1.00223234e+00),
     ( 4.51132719, -1.59454172e+00), ( 3.75317695, -1.23760511e+00),
     ( 4.00571043, -1.22269297e+00), ( 2.17587152, -6.09199048e-01),
     (-1.68565017,  4.28215858e-01), ( 1.70540789, -3.87930904e-01),
     (-4.37027024,  8.75581014e-01), ( 4.69297486, -8.10652606e-01)]>
```

Total running time of the script: (0 minutes 0.383 seconds)

`Download Python source code: plot_sgr-coordinate-frame.py`

`Download Jupyter notebook: plot_sgr-coordinate-frame.ipynb`

Gallery generated by Sphinx-Gallery

astropy.io

General examples of the `astropy.io` subpackages.

Create a multi-extension FITS (MEF) file from scratch

Note

Click here to download the full example code

Create a multi-extension FITS (MEF) file from scratch

This example demonstrates how to create a multi-extension FITS (MEF) file from scratch using `astropy.io.fits`.

By: Erik Bray

License: BSD

```
import os
```

HDUList objects are used to hold all the HDUs in a FITS file. This `HDUList` class is a subclass of Python's builtin `list`. and can be created from scratch. For example, to create a FITS file with three extensions:

```
from astropy.io import fits
new_hdul = fits.HDUList()
```

```
new_hdul.append(fits.ImageHDU())
new_hdul.append(fits.ImageHDU())
```

Write out the new file to disk:

```
new_hdul.writeto('test.fits')
```

Alternatively, the HDU instances can be created first (or read from an existing FITS file).

Create a multi-extension FITS file with two empty IMAGE extensions (a default PRIMARY HDU is prepended automatically if one is not specified; we use `overwrite=True` to overwrite the file if it already exists):

```
hdu1 = fits.PrimaryHDU()
hdu2 = fits.ImageHDU()
new_hdul = fits.HDUList([hdu1, hdu2])
new_hdul.writeto('test.fits', overwrite=True)
```

Finally, we'll remove the file we created:

```
os.remove('test.fits')
```

Total running time of the script: (0 minutes 0.005 seconds)

`Download Python source code: create-mef.py`

`Download Jupyter notebook: create-mef.ipynb`

Gallery generated by Sphinx-Gallery

Accessing data stored as a table in a multi-extension FITS (MEF) file

Note

Click here to download the full example code

Accessing data stored as a table in a multi-extension FITS (MEF) file

FITS files can often contain large amount of multi-dimensional data and tables. This example opens a FITS file with information from Chandra’s HETG-S instrument.

The example uses **astropy.utils.data** to download multi-extension FITS (MEF) file, **astropy.io.fits** to investigate the header, and **astropy.table.Table** to explore the data.

By: Lia Corrales, Adrian Price-Whelan, and Kelle Cruz

License: BSD

Use **astropy.utils.data** subpackage to download the FITS file used in this example. Also import **Table** from the **astropy.table** subpackage and **astropy.io.fits**

```
from astropy.utils.data import get_pkg_data_filename
from astropy.table import Table
from astropy.io import fits
```

Download a FITS file

```
event_filename = get_pkg_data_filename('tutorials/FITS-tables
/chandra_events.fits')
```

Display information about the contents of the FITS file.

```
fits.info(event_filename)
```

Out:

```
Filename: /home/auz/.astropy/cache/download
/url/333246bccb141ea3b4e86c49e45bf8d6/contents
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU      30   ()
  1  EVENTS        1 BinTableHDU    890   483964R x 19C   [1D, 1I, 1I,
1J, 1I, 1I, 1I, 1I, 1E, 1E, 1E, 1E, 1J, 1J, 1E, 1J, 1I, 1I, 32X]
  2  GTI           3 BinTableHDU     28   1R x 2C   [1D, 1D]
  3  GTI           2 BinTableHDU     28   1R x 2C   [1D, 1D]
  4  GTI           1 BinTableHDU     28   1R x 2C   [1D, 1D]
  5  GTI           0 BinTableHDU     28   1R x 2C   [1D, 1D]
  6  GTI           6 BinTableHDU     28   1R x 2C   [1D, 1D]
```

Extension 1, EVENTS, is a Table that contains information about each X-ray photon that hit Chandra’s HETG-S detector.

Use **Table** to read the table

```
events = Table.read(event_filename, hdu=1)
```

Print the column names of the Events Table.

```
print(events.columns)
```

Out:

```
<TableColumns names=
('time','ccd_id','node_id','expno','chipx','chipy','tdetx','tdety','det
x','dety','x','y','pha','pha_ro','energy','pi','fltgrade','grade','stat
us')>
```

If a column contains unit information, it will have an associated **astropy.units** object.

```
print(events['energy'].unit)
```

Out:

```
eV
```

Print the data stored in the Energy column.

```
print(events['energy'])
```

Out:

```
  energy
    eV
---------
13874.715
2621.1938
12119.018
3253.0364
```

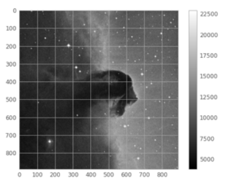
```
14214.382
1952.7239
3267.5334
3817.0366
2252.7295
6154.1094
      ...
4819.8286
12536.866
2599.5652
15535.768
6653.0815
14362.482
14653.954
 6652.827
 9672.882
1875.9359
Length = 483964 rows
```

Total running time of the script: (0 minutes 6.336 seconds)

`Download Python source code: fits-tables.py`

`Download Jupyter notebook: fits-tables.ipynb`

Gallery generated by Sphinx-Gallery



Read and plot an image from a FITS file

Note

Click here to download the full example code

Read and plot an image from a FITS file

This example opens an image stored in a FITS file and displays it to the screen.

This example uses **astropy.utils.data** to download the file, **astropy.io.fits** to open the file, and **matplotlib.pyplot** to display the image.

By: Lia R. Corrales, Adrian Price-Whelan, Kelle Cruz

License: BSD

Set up matplotlib and use a nicer set of plot parameters

```
import matplotlib.pyplot as plt
from astropy.visualization import astropy_mpl_style
plt.style.use(astropy_mpl_style)
```

Download the example FITS files used by this example:

```
from astropy.utils.data import get_pkg_data_filename
from astropy.io import fits

image_file = get_pkg_data_filename('tutorials/FITS-images
/HorseHead.fits')
```

Use **astropy.io.fits.info()** to display the structure of the file:

```
fits.info(image_file)
```

Out:

```
Filename: /home/auz/.astropy/cache/download
/url/ff6e0b93871033c68022ca026a956d87/contents
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU     161   (891, 893)   int16
  1  er.mask       1 TableHDU        25   1600R x 4C   [F6.2, F6.2,
F6.2, F6.2]
```

Generally the image information is located in the Primary HDU, also known as extension 0. Here, we use **astropy.io.fits.getdata()** to read the image data from this first extension using the keyword argument `ext=0`:

```
image_data = fits.getdata(image_file, ext=0)
```

The data is now stored as a 2D numpy array. Print the dimensions using the shape attribute:

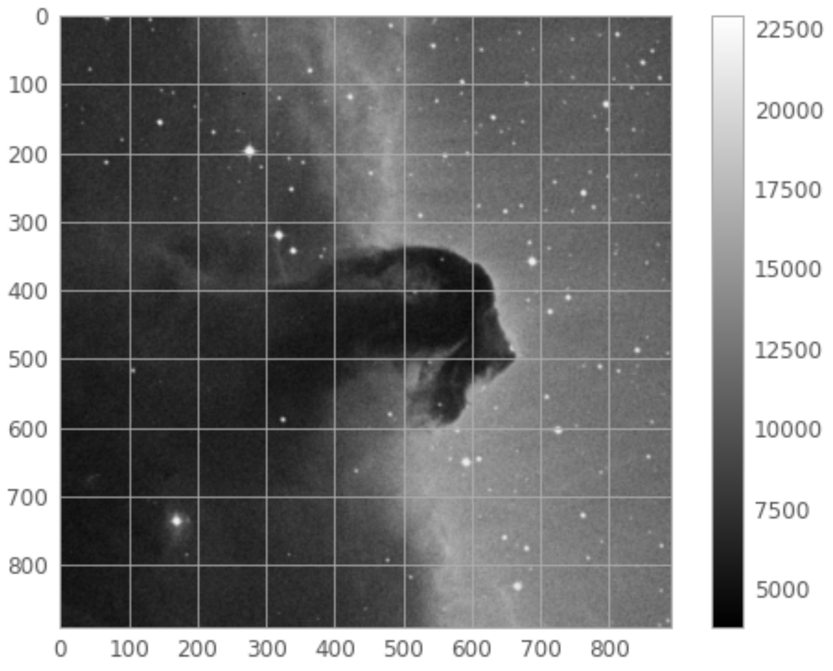
```
print(image_data.shape)
```

Out:

```
(893, 891)
```

Display the image data:

```
plt.figure()
plt.imshow(image_data, cmap='gray')
plt.colorbar()
```



Out:

```
<matplotlib.colorbar.Colorbar object at 0x7fb4c65bba00>
```

Total running time of the script: (0 minutes 2.003 seconds)

`Download Python source code: plot_fits-image.py`

Download Jupyter notebook: plot_fits-image.ipynb

Gallery generated by Sphinx-Gallery

Edit a FITS header

Note

Click here to download the full example code

Edit a FITS header

This example describes how to edit a value in a FITS header using **astropy.io.fits**.

By: Adrian Price-Whelan

License: BSD

```
from astropy.io import fits
```

Download a FITS file:

```
from astropy.utils.data import get_pkg_data_filename

fits_file = get_pkg_data_filename('tutorials/FITS-Header
/input_file.fits')
```

Look at contents of the FITS file

```
fits.info(fits_file)
```

Out:

```
Filename: /home/auz/.astropy/cache/download
/url/519010d87325a22575dc1d16f3a05d26/contents
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU       7   (100, 100)   float64
  1                 1 ImageHDU         7   (128, 128)   float64
```

Look at the headers of the two extensions:

```
print("Before modifications:")
print()
print("Extension 0:")
print(repr(fits.getheader(fits_file, 0)))
print()
print("Extension 1:")
print(repr(fits.getheader(fits_file, 1)))
```

Out:

```
Before modifications:

Extension 0:
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                  -64 / array data type
NAXIS   =                    2 / number of array dimensions
NAXIS1  =                  100
NAXIS2  =                  100
EXTEND  =                    T
OBJECT  = 'KITTEN  '

Extension 1:
XTENSION= 'IMAGE   '           / Image extension
BITPIX  =                  -64 / array data type
NAXIS   =                    2 / number of array dimensions
NAXIS1  =                  128
NAXIS2  =                  128
PCOUNT  =                    0 / number of parameters
GCOUNT  =                    1 / number of groups
```

`astropy.io.fits` provides an object-oriented interface for reading and interacting with FITS files, but for small operations (like this example) it is often easier to use the convenience functions.

To edit a single header value in the header for extension 0, use the **`setval()`** function. For example, set the OBJECT keyword to ‘M31’:

```
fits.setval(fits_file, 'OBJECT', value='M31')
```

With no extra arguments, this will modify the header for extension 0, but this can be changed using the `ext` keyword argument. For example, we can specify extension 1 instead:

```
fits.setval(fits_file, 'OBJECT', value='M31', ext=1)
```

This can also be used to create a new keyword-value pair ("card" in FITS lingo):

```
fits.setval(fits_file, 'ANEWKEY', value='some value')
```

Again, this is useful for one-off modifications, but can be inefficient for operations like editing multiple headers in the same file because **`setval()`** loads the whole file each time it is called. To make several modifications, it's better to load the file once:

```
with fits.open(fits_file, 'update') as f:
    for hdu in f:
        hdu.header['OBJECT'] = 'CAT'

print("After modifications:")
print()
print("Extension 0:")
print(repr(fits.getheader(fits_file, 0)))
print()
print("Extension 1:")
print(repr(fits.getheader(fits_file, 1)))
```

Out:

```
After modifications:

Extension 0:
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                  -64 / array data type
NAXIS   =                    2 / number of array dimensions
NAXIS1  =                  100
NAXIS2  =                  100
EXTEND  =                    T
OBJECT  = 'CAT     '
ANEWKEY = 'some value'

Extension 1:
XTENSION= 'IMAGE   '           / Image extension
BITPIX  =                  -64 / array data type
NAXIS   =                    2 / number of array dimensions
NAXIS1  =                  128
```

```
NAXIS2  =                  128
PCOUNT  =                    0 / number of parameters
GCOUNT  =                    1 / number of groups
OBJECT  = 'CAT     '
```

Total running time of the script: (0 minutes 0.767 seconds)

Download Python source code: modify-fits-header.py

Download Jupyter notebook: modify-fits-header.ipynb

Gallery generated by Sphinx-Gallery

Create a very large FITS file from scratch

Note

Click here to download the full example code

Create a very large FITS file from scratch

This example demonstrates how to create a large file (larger than will fit in memory) from scratch using `astropy.io.fits`.

By: Erik Bray

License: BSD

Normally to create a single image FITS file one would do something like:

```python
import os
import numpy as np
from astropy.io import fits
data = np.zeros((40000, 40000), dtype=np.float64)
hdu = fits.PrimaryHDU(data=data)
```

Then use the **astropy.io.fits.writeto()** method to write out the new file to disk

```
hdu.writeto('large.fits')
```

However, a 40000 x 40000 array of doubles is nearly twelve gigabytes! Most systems won’t be able to create that in memory just to write out to disk. In order to create such a large file efficiently requires a little extra work, and a few assumptions.

First, it is helpful to anticipate about how large (as in, how many keywords) the header will have in it. FITS headers must be written in 2880 byte blocks, large enough for 36 keywords per block (including the END keyword in the final block). Typical headers have somewhere between 1 and 4 blocks, though sometimes more.

Since the first thing we write to a FITS file is the header, we want to write enough header blocks so that there is plenty of padding in which to add new keywords without having to resize the whole file. Say you want the header to use 4 blocks by default. Then, excluding the END card which Astropy will add automatically, create the header and pad it out to 36 * 4 cards.

Create a stub array to initialize the HDU; its exact size is irrelevant, as long as it has the desired number of dimensions

```
data = np.zeros((100, 100), dtype=np.float64)
hdu = fits.PrimaryHDU(data=data)
header = hdu.header
while len(header) < (36 * 4 - 1):
    header.append()  # Adds a blank card to the end
```

Now adjust the NAXISn keywords to the desired size of the array, and write only the header out to a file. Using the `hdu.writeto()` method will cause astropy to “helpfully” reset the NAXISn keywords to match the size of the dummy array. That is because it works hard to ensure that only valid FITS files are written. Instead, we can write just the header to a file using the **astropy.io.fits.Header.tofile** method:

```
header['NAXIS1'] = 40000
header['NAXIS2'] = 40000
header.tofile('large.fits')
```

Finally, grow out the end of the file to match the length of the data (plus the length of the header). This can be done very efficiently on most systems by seeking past the end of the file and writing a single byte, like so:

```
with open('large.fits', 'rb+') as fobj:
    # Seek past the length of the header, plus the length of the
    # Data we want to write.
    # 8 is the number of bytes per value, i.e.
abs(header['BITPIX'])/8
    # (this example is assuming a 64-bit float)
    # The -1 is to account for the final byte that we are about to
    # write:
    fobj.seek(len(header.tostring()) + (40000 * 40000 * 8) - 1)
    fobj.write(b'\0')
```

More generally, this can be written:

```
shape = tuple(header['NAXIS{0}'.format(ii)] for ii in range(1,
header['NAXIS']+1))
with open('large.fits', 'rb+') as fobj:
    fobj.seek(len(header.tostring()) + (np.product(shape) *
np.abs(header['BITPIX']//8)) - 1)
    fobj.write(b'\0')
```

On modern operating systems this will cause the file (past the header) to be filled with zeros out to the ~12GB needed to hold a 40000 x 40000 image. On filesystems that support sparse file creation (most Linux filesystems, but not the HFS+ filesystem used by most Macs) this is a very fast, efficient operation. On other systems your mileage may vary.

This isn't the only way to build up a large file, but probably one of the safest. This method can also be used to create large multi-extension FITS files, with a little care.

Finally, we'll remove the file we created:

```
os.remove('large.fits')
```

Total running time of the script: (0 minutes 0.000 seconds)

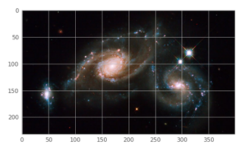
`Download Python source code: skip_create-large-fits.py`

`Download Jupyter notebook: skip_create-large-fits.ipynb`

Gallery generated by Sphinx-Gallery

Note

Click here to download the full example code



Convert a 3-color image (JPG) to separate FITS images

Convert a 3-color image (JPG) to separate FITS images

This example opens an RGB JPEG image and writes out each channel as a separate FITS (image) file.

This example uses pillow to read the image, **`matplotlib.pyplot`** to display the image, and **`astropy.io.fits`** to save FITS files.

By: Erik Bray, Adrian Price-Whelan

License: BSD

```
import numpy as np
from PIL import Image
from astropy.io import fits
```

Set up matplotlib and use a nicer set of plot parameters

```
import matplotlib.pyplot as plt
from astropy.visualization import astropy_mpl_style
plt.style.use(astropy_mpl_style)
```

Load and display the original 3-color jpeg image:

```
image = Image.open('Hs-2009-14-a-web.jpg')
xsize, ysize = image.size
print(f"Image size: {ysize} x {xsize}")
print(f"Image bands: {image.getbands()}")
ax = plt.imshow(image)
```

Out:

```
Image size: 232 x 400
Image bands: ('R', 'G', 'B')
```

Split the three channels (RGB) and get the data as Numpy arrays. The arrays are flattened, so they are 1-dimensional:

```
r, g, b = image.split()
r_data = np.array(r.getdata()) # data is now an array of length
ysize*xsize
g_data = np.array(g.getdata())
b_data = np.array(b.getdata())
print(r_data.shape)
```

Out:

```
(92800,)
```

Reshape the image arrays to be 2-dimensional:

```
r_data = r_data.reshape(ysize, xsize) # data is now a matrix (ysize,
xsize)
g_data = g_data.reshape(ysize, xsize)
b_data = b_data.reshape(ysize, xsize)
print(r_data.shape)
```

Out:

```
(232, 400)
```

Write out the channels as separate FITS images. Add and visualize header info

```
red = fits.PrimaryHDU(data=r_data)
```

```
red.header['LATOBS'] = "32:11:56" # add spurious header info
red.header['LONGOBS'] = "110:56"
red.writeto('red.fits')

green = fits.PrimaryHDU(data=g_data)
green.header['LATOBS'] = "32:11:56"
green.header['LONGOBS'] = "110:56"
green.writeto('green.fits')

blue = fits.PrimaryHDU(data=b_data)
blue.header['LATOBS'] = "32:11:56"
blue.header['LONGOBS'] = "110:56"
blue.writeto('blue.fits')

from pprint import pprint
pprint(red.header)
```

Out:

```
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                   64 / array data type
NAXIS   =                    2 / number of array dimensions
NAXIS1  =                  400
NAXIS2  =                  232
EXTEND  =                    T
LATOBS  = '32:11:56'
LONGOBS = '110:56  '
```

Delete the files created

```
import os
os.remove('red.fits')
os.remove('green.fits')
os.remove('blue.fits')
```

Total running time of the script: (0 minutes 0.143 seconds)

Download Python source code: split-jpeg-to-fits.py

Download Jupyter notebook: split-jpeg-to-fits.ipynb

Gallery generated by Sphinx-Gallery

Download all examples in Python source code: examples_python.zip

Download all examples in Jupyter notebooks: examples_jupyter.zip

Gallery generated by Sphinx-Gallery

User Documentation

Data structures and transformations

Constants (`astropy.constants`)

Introduction

`astropy.constants` contains a number of physical constants useful in Astronomy. Constants are `Quantity` objects with additional metadata describing their provenance and uncertainties.

Getting Started

To use the constants in International System of Units (SI units), you can import the constants directly from the `astropy.constants` sub-package:

```
>>> from astropy.constants import G
```

Or, if you want to avoid having to explicitly import all of the constants you need, you can do:

```
>>> from astropy import constants as const
```

and then subsequently use, for example, `const.G`. Constants are fully-fledged `Quantity` objects, so you can conveniently convert them to different units.

Example

To convert constants to different units:

```
>>> print(const.c)
```

```
  Name   = Speed of light in vacuum
  Value  = 299792458.0
  Uncertainty  = 0.0
  Unit  = m / s
  Reference = CODATA 2018

>>> print(const.c.to('km/s'))
299792.458 km / s

>>> print(const.c.to('pc/yr'))
0.306601393788 pc / yr
```

You can then use them in conjunction with unit and other nonconstant **`Quantity`** objects:

```
>>> from astropy import units as u
>>> F = (const.G * 3. * const.M_sun * 100 * u.kg) / (2.2 * u.au) ** 2
>>> print(F.to(u.N))
0.3675671602160826 N
```

It is possible to convert most constants to Centimeter-Gram-Second (CGS) units using, for example:

```
>>> const.c.cgs
<Quantity   2.99792458e+10 cm / s>
```

However, some constants are defined with different physical dimensions in CGS and cannot be directly converted. Because of this ambiguity, such constants cannot be used in expressions without specifying a system:

```
>>> 100 * const.e
Traceback (most recent call last):
    ...
TypeError: Constant u'e' does not have physically compatible units
across all systems of units and cannot be combined with other
values without specifying a system (eg. e.emu)
>>> 100 * const.e.esu
<Quantity 4.8032045057134676e-08 Fr>
```

Collections of Constants (and Prior Versions)

Constants are organized into version modules. The constants for `astropy` 2.0 can be accessed in the `astropyconst20` module. For example:

```
>>> from astropy.constants import astropyconst20 as const
>>> print(const.e)
```

```
  Name   = Electron charge
  Value  = 1.6021766208e-19
  Uncertainty  = 9.8e-28
  Unit  = C
  Reference = CODATA 2014
```

Physical CODATA constants are in modules with names like `codata2010`, `codata2014`, or `codata2018`:

```
>>> from astropy.constants import codata2014 as const
>>> print(const.h)
  Name   = Planck constant
  Value  = 6.62607004e-34
  Uncertainty  = 8.1e-42
  Unit  = J s
  Reference = CODATA 2014
```

Astronomical constants defined (primarily) by the International Astronomical Union (IAU) are collected in modules with names like `iau2012` or `iau2015`:

```
>>> from astropy.constants import iau2012 as const
>>> print(const.L_sun)
  Name   = Solar luminosity
  Value  = 3.846e+26
  Uncertainty  = 5e+22
  Unit  = W
  Reference = Allen's Astrophysical Quantities 4th Ed.
```

```
>>> from astropy.constants import iau2015 as const
>>> print(const.L_sun)
  Name   = Nominal solar luminosity
  Value  = 3.828e+26
  Uncertainty  = 0.0
  Unit  = W
  Reference = IAU 2015 Resolution B 3
```

The astronomical and physical constants are combined into modules with names like `astropyconst13`, `astropyconst20`, and `astropyconst40` for different versions. However, importing these prior version modules directly will lead to inconsistencies with other subpackages that have already imported **`astropy.constants`**. Notably, **`astropy.units`** will have already used the default version of constants. When using prior versions of the constants in this manner, quantities should be constructed with constants instead of units.

To ensure consistent use of a prior version of constants in other Astropy packages (such as **astropy.units**) that import `constants`, the physical and astronomical constants versions should be set via `ScienceState` classes. These must be set before the first import of either **astropy.constants** or **astropy.units**. For example, you can use the CODATA2010 physical constants and the IAU 2012 astronomical constants:

```
>>> from astropy import physical_constants, astronomical_constants
>>> physical_constants.set('codata2010')
<ScienceState physical_constants: 'codata2010'>
>>> physical_constants.get()
'codata2010'
>>> astronomical_constants.set('iau2012')
<ScienceState astronomical_constants: 'iau2012'>
>>> astronomical_constants.get()
'iau2012'
```

Then all other packages that import **astropy.constants** will self-consistently initialize with that prior version of constants.

The versions may also be set using values referring to the version modules:

```
>>> from astropy import physical_constants, astronomical_constants
>>> physical_constants.set('astropyconst13')
<ScienceState physical_constants: 'codata2010'>
>>> physical_constants.get()
'codata2010'
>>> astronomical_constants.set('astropyconst13')
<ScienceState astronomical_constants: 'iau2012'>
>>> astronomical_constants.get()
'iau2012'
```

If either **astropy.constants** or **astropy.units** have already been imported, a `RuntimeError` will be raised.

```
>>> import astropy.units
>>> from astropy import physical_constants, astronomical_constants
>>> astronomical_constants.set('astropyconst13')
Traceback (most recent call last):
    ...
RuntimeError: astropy.units is already imported
```

Reference/API

astropy.constants Package

Contains astronomical and physical constants for use in Astropy or other places.

A typical use case might be:

```
>>> from astropy.constants import c, m_e
>>> # ... define the mass of something you want the rest energy of as
m ...
>>> m = m_e
>>> E = m * c**2
>>> E.to('MeV')
<Quantity 0.510998927603161 MeV>
```

The following constants are available:

Name	Value	Unit	Description
G	6.6743e-11	m3 / (kg s2)	Gravitational constant
N_A	6.02214076e+23	1 / (mol)	Avogadro's number
R	8.31446262	J / (K mol)	Gas constant
Ryd	10973731.6	1 / (m)	Rydberg constant
a0	5.29177211e-11	m	Bohr radius
alpha	0.00729735257		Fine-structure constant
atm	101325	Pa	Standard atmosphere
b_wien	0.00289777196	m K	Wien wavelength displacement law constant
c	299792458	m / (s)	Speed of light in vacuum
e	1.60217663e-19	C	Electron charge
eps0	8.85418781e-12	F/m	Vacuum electric permittivity
g0	9.80665	m / s2	Standard acceleration of gravity
h	6.62607015e-34	J s	Planck constant
hbar	1.05457182e-34	J s	Reduced Planck constant
k_B	1.380649e-23	J / (K)	Boltzmann constant
m_e	9.1093837e-31	kg	Electron mass
m_n	1.6749275e-27	kg	Neutron mass
m_p	1.67262192e-27	kg	Proton mass
mu0	1.25663706e-06	N/A2	Vacuum magnetic permeability
muB	9.27401008e-24	J/T	Bohr magneton
sigma_T	6.65245873e-29	m2	Thomson scattering cross-section
sigma_sb	5.67037442e-08	W / (K4 m2)	Stefan-Boltzmann constant
u	1.66053907e-27	kg	Atomic mass

Name	Value	Unit	Description
GM_earth	3.986004e+14	m3 / (s2)	Nominal Earth mass parameter
GM_jup	1.2668653e+17	m3 / (s2)	Nominal Jupiter mass parameter
GM_sun	1.3271244e+20	m3 / (s2)	Nominal solar mass parameter
L_bol0	3.0128e+28	W	Luminosity for absolute bolometric magnitude 0
L_sun	3.828e+26	W	Nominal solar luminosity
M_earth	5.97216787e+24	kg	Earth mass
M_jup	1.8981246e+27	kg	Jupiter mass
M_sun	1.98840987e+30	kg	Solar mass
R_earth	6378100	m	Nominal Earth equatorial radius
R_jup	71492000	m	Nominal Jupiter equatorial radius
R_sun	695700000	m	Nominal solar radius
au	1.49597871e+11	m	Astronomical Unit
kpc	3.08567758e+19	m	Kiloparsec
pc	3.08567758e+16	m	Parsec

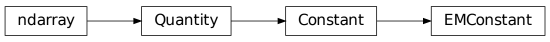
Functions

`set_enabled_constants`(modname)	● ● *Deprecated since version 4.0.*

Classes

`Constant`(abbrev, name, value, unit, uncertainty)	A physical or astronomical constant.
`EMConstant`(abbrev, name, value, unit, …[, …])	An electromagnetic constant.

Class Inheritance Diagram



Units and Quantities (astropy.units)

Introduction

astropy.units handles defining, converting between, and performing arithmetic with physical quantities, such as meters, seconds, Hz, etc. It also handles logarithmic units such as magnitude and decibel.

astropy.units does not know spherical geometry or sexagesimal (hours, min, sec): if you want to deal with celestial coordinates, see the astropy.coordinates package.

Getting Started

Most users of the astropy.units package will work with Quantity objects: the combination of a value and a unit. The most convenient way to create a Quantity is to multiply or divide a value by one of the built-in units. It works with scalars, sequences, and `numpy` arrays.

Examples

To create a Quantity object:

```
>>> from astropy import units as u
>>> 42.0 * u.meter
<Quantity  42. m>
>>> [1., 2., 3.] * u.m
<Quantity [1., 2., 3.] m>
>>> import numpy as np
>>> np.array([1., 2., 3.]) * u.m
<Quantity [1., 2., 3.] m>
```

You can get the unit and value from a Quantity using the unit and value members:

```
>>> q = 42.0 * u.meter
>>> q.value
42.0
>>> q.unit
```

```
Unit("m")
```

From this basic building block, it is possible to start combining quantities with different units:

```
>>> 15.1 * u.meter / (32.0 * u.second)
<Quantity 0.471875 m / s>
>>> 3.0 * u.kilometer / (130.51 * u.meter / u.second)
<Quantity 0.022986744310780783 km s / m>
>>> (3.0 * u.kilometer / (130.51 * u.meter / u.second)).decompose()
<Quantity 22.986744310780782 s>
```

Unit conversion is done using the `to()` method, which returns a new `Quantity` in the given unit:

```
>>> x = 1.0 * u.parsec
>>> x.to(u.km)
<Quantity 30856775814671.914 km>
```

It is also possible to work directly with units at a lower level, for example, to create custom units:

```
>>> from astropy.units import imperial

>>> cms = u.cm / u.s
>>> # ...and then use some imperial units
>>> mph = imperial.mile / u.hour

>>> # And do some conversions
>>> q = 42.0 * cms
>>> q.to(mph)
<Quantity 0.939513242662849 mi / h>
```

Units that “cancel out” become a special unit called the “dimensionless unit”:

```
>>> u.m / u.m
Unit(dimensionless)
```

To create a basic dimensionless quantity, multiply a value by the unscaled dimensionless unit:

```
>>> q = 1.0 * u.dimensionless_unscaled
>>> q.unit
Unit(dimensionless)
```

astropy.units is able to match compound units against the units it already knows about:

```
>>> (u.s ** -1).compose()
[Unit("Bq"), Unit("Hz"), Unit("3.7e+10 Ci")]
```

And it can convert between unit systems, such as SI or CGS:

```
>>> (1.0 * u.Pa).cgs
<Quantity 10.0 Ba>
```

The units `mag`, `dex`, and `dB` are special, being logarithmic units, for which a value is the logarithm of a physical quantity in a given unit. These can be used with a physical unit in parentheses to create a corresponding logarithmic quantity:

```
>>> -2.5 * u.mag(u.ct / u.s)
<Magnitude -2.5 mag(ct / s)>
>>> from astropy import constants as c
>>> u.Dex((c.G * u.M_sun / u.R_sun**2).cgs)
<Dex 4.438067627303133 dex(cm / s2)>
```

astropy.units also handles equivalencies, such as that between wavelength and frequency. To use that feature, equivalence objects are passed to the **to()** conversion method. For instance, a conversion from wavelength to frequency does not normally work:

```
>>> (1000 * u.nm).to(u.Hz)
Traceback (most recent call last):
  ...
UnitConversionError: 'nm' (length) and 'Hz' (frequency) are not
convertible
```

But by passing an equivalency list, in this case `spectral()`, it does:

```
>>> (1000 * u.nm).to(u.Hz, equivalencies=u.spectral())
<Quantity  2.99792458e+14 Hz>
```

Quantities and units can be printed nicely to strings using the Format String Syntax. Format specifiers (like `0.03f`) in strings will be used to format the quantity value:

```
>>> q = 15.1 * u.meter / (32.0 * u.second)
>>> q
```

```
<Quantity 0.471875 m / s>
>>> f"{q:0.03f}"
'0.472 m / s'
```

The value and unit can also be formatted separately. Format specifiers for units can be used to choose the unit formatter:

```
>>> q = 15.1 * u.meter / (32.0 * u.second)
>>> q
<Quantity 0.471875 m / s>
>>> f"{q.value:0.03f} {q.unit:FITS}"
'0.472 m s-1'
```

Using `astropy.units`

Quantity

The `Quantity` object is meant to represent a value that has some unit associated with the number.

Creating Quantity Instances

`Quantity` objects are normally created through multiplication with `Unit` objects.

Examples

To create a `Quantity` to represent 15 m/s:

```
>>> import astropy.units as u
>>> 15 * u.m / u.s
<Quantity 15. m / s>
```

This extends as expected to division by a unit, or using `numpy` arrays or Python sequences:

```
>>> 1.25 / u.s
<Quantity 1.25 1 / s>
>>> [1, 2, 3] * u.m
<Quantity [1., 2., 3.] m>
>>> import numpy as np
>>> np.array([1, 2, 3]) * u.m
<Quantity [1., 2., 3.] m>
```

You can also create instances using the **`Quantity`** constructor directly, by specifying a value and unit:

```
>>> u.Quantity(15, u.m / u.s)
<Quantity 15. m / s>
```

The constructor gives a few more options. In particular, it allows you to merge sequences of **`Quantity`** objects (as long as all of their units are equivalent), and to parse simple strings (which may help, for example, to parse configuration files, etc.):

```
>>> qlst = [60 * u.s, 1 * u.min]
>>> u.Quantity(qlst, u.minute)
<Quantity [1.,  1.] min>
>>> u.Quantity('15 m/s')
<Quantity 15. m / s>
```

The current unit and value can be accessed via the **`unit`** and **`value`** attributes:

```
>>> q = 2.5 * u.m / u.s
>>> q.unit
Unit("m / s")
>>> q.value
2.5
```

Note

`Quantity` objects are converted to float by default. Furthermore, any data passed in are copied, which for large arrays may not be optimal. As discussed further below, you can instead obtain a **`view`** by passing `copy=False` to **`Quantity`** or use the `<<` operator.

Converting to Different Units

`Quantity` objects can be converted to different units using the **`to()`** method.

Examples

To convert **`Quantity`** objects to different units:

```
>>> q = 2.3 * u.m / u.s
>>> q.to(u.km / u.h)
```

```
<Quantity 8.28 km / h>
```

For convenience, the `si` and `cgs` attributes can be used to convert the `Quantity` to base SI or CGS units:

```
>>> q = 2.4 * u.m / u.s
>>> q.si
<Quantity 2.4 m / s>
>>> q.cgs
<Quantity 240. cm / s>
```

If you want the value of the quantity in a different unit, you can use `to_value()` as a shortcut:

```
>>> q = 2.5 * u.m
>>> q.to_value(u.cm)
250.0
```

Note

You could get the value in `cm` also using `q.to(u.cm).value`. The difference is that `to_value()` does no conversion if the unit is already the correct one, instead returning an `view()` of the data (just as if you had done `q.value`). In contrast, `to()` always returns a copy (which also means it is slower for the case where no conversion is necessary). As discussed further below, you can avoid the copy if the unit is already correct by using the `<<` operator.

Comparing Quantities

`Quantity` objects can be compared as follows:

```
>>> from astropy import units as u
>>> u.allclose([1, 2] * u.m, [100, 200] * u.cm)
True
>>> u.isclose([1, 2] * u.m, [100, 20] * u.cm)
array([ True, False])
```

Plotting Quantities

`Quantity` objects can be conveniently plotted using matplotlib — see Plotting quantities for more details.

Arithmetic

Addition and Subtraction

Addition or subtraction between **`Quantity`** objects is supported when their units are equivalent.

Examples

When the units are equal, the resulting object has the same unit:

```
>>> 11 * u.s + 30 * u.s
<Quantity 41. s>
>>> 30 * u.s - 11 * u.s
<Quantity 19. s>
```

If the units are equivalent, but not equal (e.g., kilometer and meter), the resulting object **has units of the object on the left**:

```
>>> 1100.1 * u.m + 13.5 * u.km
<Quantity 14600.1 m>
>>> 13.5 * u.km + 1100.1 * u.m
<Quantity 14.6001 km>
>>> 1100.1 * u.m - 13.5 * u.km
<Quantity -12399.9 m>
>>> 13.5 * u.km - 1100.1 * u.m
<Quantity 12.3999 km>
```

Addition and subtraction are not supported between **`Quantity`** objects and basic numeric types:

```
>>> 13.5 * u.km + 19.412
Traceback (most recent call last):
  ...
UnitConversionError: Can only apply 'add' function to dimensionless
quantities when other argument is not a quantity (unless the
latter is all zero/infinity/nan)
```

Except for dimensionless quantities (see Dimensionless Quantities).

Multiplication and Division

Multiplication and division are supported between **`Quantity`** objects with any units, and with numeric types. For these operations between objects with equivalent units, the **resulting object has composite units**.

Examples

To perform these operations on **`Quantity`** objects:

```
>>> 1.1 * u.m * 140.3 * u.cm
<Quantity 154.33 cm m>
>>> 140.3 * u.cm * 1.1 * u.m
<Quantity 154.33 cm m>
>>> 1. * u.m / (20. * u.cm)
<Quantity 0.05 m / cm>
>>> 20. * u.cm / (1. * u.m)
<Quantity 20. cm / m>
```

For multiplication, you can change how to represent the resulting object by using the **`to()`** method:

```
>>> (1.1 * u.m * 140.3 * u.cm).to(u.m**2)
<Quantity 1.5433 m2>
>>> (1.1 * u.m * 140.3 * u.cm).to(u.cm**2)
<Quantity 15433. cm2>
```

For division, if the units are equivalent, you may want to make the resulting object dimensionless by reducing the units. To do this, use the **`decompose()`** method:

```
>>> (20. * u.cm / (1. * u.m)).decompose()
<Quantity 0.2>
```

This method is also useful for more complicated arithmetic:

```
>>> 15. * u.kg * 32. * u.cm * 15 * u.m / (11. * u.s * 1914.15 * u.ms)
<Quantity 0.34195097 cm kg m / (ms s)>
>>> (15. * u.kg * 32. * u.cm * 15 * u.m / (11. * u.s * 1914.15 *
u.ms)).decompose()
<Quantity 3.41950973 kg m2 / s2>
```

NumPy Functions

`Quantity` objects are actually full `numpy` arrays (the **`Quantity`** class inherits from and extends **`numpy.ndarray`**), and we have tried to ensure that `numpy` functions behave properly with quantities:

```
>>> q = np.array([1., 2., 3., 4.]) * u.m / u.s
>>> np.mean(q)
<Quantity 2.5 m / s>
>>> np.std(q)
<Quantity 1.11803399 m / s>
```

This includes functions that only accept specific units such as angles:

```
>>> q = 30. * u.deg
>>> np.sin(q)
<Quantity 0.5>
```

Or dimensionless quantities:

```
>>> from astropy.constants import h, k_B
>>> nu = 3 * u.GHz
>>> T = 30 * u.K
>>> np.exp(-h * nu / (k_B * T))
<Quantity 0.99521225>
```

See Dimensionless Quantities below for more details.

Note

With `numpy` versions older than 1.17, a number of mostly non-arithmetic functions have known issues, either ignoring the unit (e.g., `np.dot`) or not reinitializing it properly (e.g., `np.hstack`). This propagates to more complex functions such as `np.linalg.norm`.

Support for functions from other packages, such as `scipy`, is more incomplete (contributions to improve this welcomed!).

Dimensionless Quantities

Dimensionless quantities have the characteristic that if they are added or subtracted from a Python scalar or unitless **`ndarray`**, or if they are passed to a `numpy` function that takes dimensionless quantities, the units are simplified so that the quantity is dimensionless and scale-free. For example:

```
>>> 1. + 1. * u.m / u.km
<Quantity 1.001>
```

Which is different from:

```
>>> 1. + (1. * u.m / u.km).value
2.0
```

In the latter case, the result is `2.0` because the unit of `(1. * u.m / u.km)` is not scale-free by default:

```
>>> q = (1. * u.m / u.km)
>>> q.unit
Unit("m / km")
>>> q.unit.decompose()
Unit(dimensionless with a scale of 0.001)
```

However, when combining with an object that is not a **`Quantity`**, the unit is automatically decomposed to be scale-free, giving the expected result.

This also occurs when passing dimensionless quantities to functions that take dimensionless quantities:

```
>>> nu = 3 * u.GHz
>>> T = 30 * u.K
>>> np.exp(- h * nu / (k_B * T))
<Quantity 0.99521225>
```

The result is independent from the units in which the different quantities were specified:

```
>>> nu = 3.e9 * u.Hz
>>> T = 30 * u.K
>>> np.exp(- h * nu / (k_B * T))
<Quantity 0.99521225>
```

Converting to Plain Python Scalars

Converting **`Quantity`** objects does not work for non-dimensionless quantities:

```
>>> float(3. * u.m)
Traceback (most recent call last):
  ...
TypeError: only dimensionless scalar quantities can be converted
to Python scalars
```

Instead, only dimensionless values can be converted to plain Python scalars:

```
>>> float(3. * u.m / (4. * u.m))
0.75
>>> float(3. * u.km / (4. * u.m))
750.0
>>> int(6. * u.km / (2. * u.m))
3000
```

Functions that Accept Quantities

Validation of quantity arguments to functions can lead to many repetitions of the same checking code. A decorator is provided which verifies that certain arguments to a function are **Quantity** objects and that the units are compatible with a desired unit or physical type.

The decorator does not convert the input quantity to the desired unit, say arcseconds to degrees in the example below, it merely checks that such a conversion is possible, thus verifying that the **Quantity** argument can be used in calculations.

The decorator **quantity_input** accepts keyword arguments to specify which arguments should be validated and what unit they are expected to be compatible with.

Examples

To verify if a **Quantity** argument can be used in calculations:

```
>>> @u.quantity_input(myarg=u.deg)
... def myfunction(myarg):
...     return myarg.unit
```

```
>>> myfunction(100*u.arcsec)
Unit("arcsec")
```

It is also possible to instead specify the physical type of the desired unit:

```
>>> @u.quantity_input(myarg='angle')
... def myfunction(myarg):
...     return myarg.unit
```

```
>>> myfunction(100*u.arcsec)
Unit("arcsec")
```

Optionally, `None` keyword arguments are also supported; for such cases, the input is only checked when a value other than `None` is passed:

```
>>> @u.quantity_input(a='length', b='angle')
... def myfunction(a, b=None):
...     return a, b
```

```
>>> myfunction(1.*u.km)
(<Quantity 1. km>, None)
>>> myfunction(1.*u.km, 1*u.deg)
(<Quantity 1. km>, <Quantity 1. deg>)
```

Alternatively, you can use the annotations syntax to provide the units:

```
>>> @u.quantity_input
... def myfunction(myarg: u.arcsec):
...     return myarg.unit
```

```
>>> myfunction(100*u.arcsec)
Unit("arcsec")
```

You can also annotate for different types in non-unit expecting arguments:

```
>>> @u.quantity_input
... def myfunction(myarg: u.arcsec, nice_string: str):
...     return myarg.unit, nice_string
>>> myfunction(100*u.arcsec, "a nice string")
(Unit("arcsec"), 'a nice string')
```

You can define a return decoration, to which the return value will be converted, for example:

```
>>> @u.quantity_input
... def myfunction(myarg: u.arcsec) -> u.deg:
...     return myarg*1000

>>> myfunction(100*u.arcsec)
<Quantity 27.77777778 deg>
```

This both checks that the return value of your function is consistent with what you expect and makes it much neater to display the results of the function.

The decorator also supports specifying a list of valid equivalent units or physical types for functions that should accept inputs with multiple valid units:

```
>>> @u.quantity_input(a=['length', 'speed'])
... def myfunction(a):
...     return a.unit
```

```
>>> myfunction(1.*u.km)
Unit("km")
>>> myfunction(1.*u.km/u.s)
Unit("km / s")
```

Representing Vectors with Units

`Quantity` objects can, like `numpy` arrays, be used to represent vectors or matrices by assigning specific dimensions to represent the coordinates or matrix elements, but that implies tracking those dimensions carefully. For vectors, you can use instead the representations underlying coordinates, which allows you to use representations other than Cartesian (such as spherical or cylindrical), as well as simple vector arithmetic. For details, see Using and Designing Coordinate Representations.

Creating and Converting Quantities without Copies

When creating a **`Quantity`** using multiplication with a unit, a copy of the underlying data is made. This can be avoided by passing on `copy=False` in the initializer.

Examples

To avoid duplication using `copy=False`:

```
>>> a = np.arange(5.)
>>> q = u.Quantity(a, u.m, copy=False)
>>> q
<Quantity [0., 1., 2., 3., 4.] m>
>>> np.may_share_memory(a, q)
True
>>> a[0] = -1.
>>> q
<Quantity [-1.,  1.,  2.,  3.,  4.] m>
```

This may be particularly useful in functions which do not change their input; it also ensures that if a user passes in a **`Quantity`** with units of length, it will be converted to meters.

As a shortcut, you can “shift” to the requested unit using the `<<` operator:

```
>>> q = a << u.m
>>> np.may_share_memory(a, q)
True
>>> q
<Quantity [-1.,  1.,  2.,  3.,  4.] m>
```

The operator works identically to the initialization with `copy=False` mentioned above:

```
>>> q << u.cm
<Quantity [-100.,  100.,  200.,  300.,  400.] cm>
```

It can also be used for in-place conversion:

```
>>> q <<= u.cm
>>> q
<Quantity [-100.,  100.,  200.,  300.,  400.] cm>
>>> a
array([-100.,  100.,  200.,  300.,  400.])
```

Subclassing Quantity

To subclass **`Quantity`**, you generally proceed as you would when subclassing **`ndarray`** (i.e., you typically need to override `__new__`, rather than `__init__`, and use the `numpy.ndarray.__array_finalize__` method to update attributes). For details, see the NumPy documentation on subclassing. To get a sense of what is involved, have a look at **`Quantity`** itself, where, for example, the `astropy.units.Quantity.__array_finalize__` method is used to pass on the `unit`, at **`Angle`**, where strings are parsed as angles in the `astropy.coordinates.Angle.__new__` method and at **`Longitude`**, where the `astropy.coordinates.Longitude.__array_finalize__` method is used to pass on the angle at which longitudes wrap.

Another method that is meant to be overridden by subclasses, specific to **`Quantity`**, is `astropy.units.Quantity.__quantity_subclass__`. This is called to decide which type of subclass to return, based on the unit of the **`Quantity`** that is to be created. It is used, for example, in **`Angle`** to return a

`Quantity` if a calculation returns a unit other than an angular one. The implementation of this is via `SpecificTypeQuantity`, which more generally allows users to construct `Quantity` subclasses that have methods that are useful only for a specific physical type.

Standard Units

Standard units are defined in the `astropy.units` package as object instances.

All units are defined in terms of basic “irreducible” units. The irreducible units include:

- Length (meter)
- Time (second)
- Mass (kilogram)
- Current (ampere)
- Temperature (Kelvin)
- Angular distance (radian)
- Solid angle (steradian)
- Luminous intensity (candela)
- Stellar magnitude (mag)
- Amount of substance (mole)
- Photon count (photon)

(There are also some more obscure base units required by the FITS standard that are no longer recommended for use.)

Units that involve combinations of fundamental units are instances of `CompositeUnit`. In most cases, you do not need to worry about the various kinds of unit classes unless you want to design a more complex case.

There are many units already predefined in the module. You may use the `find_equivalent_units` method to list all of the existing predefined units of a given type:

```
>>> from astropy import units as u
>>> u.g.find_equivalent_units()
  Primary name | Unit definition | Aliases
[
  M_e          | 9.10938e-31 kg  |                                      ,
  M_p          | 1.67262e-27 kg  |                                      ,
  earthMass    | 5.97217e+24 kg  | M_earth, Mearth                      ,
  g            | 0.001 kg        | gram                                 ,
  jupiterMass  | 1.89812e+27 kg  | M_jup, Mjup, M_jupiter, Mjupiter ,
  kg           | irreducible     | kilogram                             ,
  solMass      | 1.98841e+30 kg  | M_sun, Msun                          ,
  t            | 1000 kg         | tonne
  u            | 1.66054e-27 kg  | Da, Dalton                          ,
]
```

Prefixes

Most units can be used with prefixes, with both the standard SI prefixes and the IEEE 1514 binary prefixes (for `bit` and `byte`) supported:

Available decimal prefixes

Symbol	Prefix	Value
Y	yotta-	1e24
Z	zetta-	1e21
E	exa-	1e18
P	peta-	1e15
T	tera-	1e12
G	giga-	1e9
M	mega-	1e6
k	kilo-	1e3
h	hecto-	1e2
da	deka-, deca	1e1
d	deci-	1e-1
c	centi-	1e-2
m	milli-	1e-3
u	micro-	1e-6
n	nano-	1e-9
p	pico-	1e-12
f	femto-	1e-15
a	atto-	1e-18
z	zepto-	1e-21
y	yocto-	1e-24

Available binary prefixes

Symbol	Prefix	Value
Ki	kibi-	2 ** 10

Available binary prefixes

Symbol	Prefix	Value
Mi	mebi-	2 ** 20
Gi	gibi-	2 ** 30
Ti	tebi-	2 ** 40
Pi	pebi-	2 ** 50
Ei	exbi-	2 ** 60

The Dimensionless Unit

In addition to these units, `astropy.units` includes the concept of the dimensionless unit, used to indicate quantities that do not have a physical dimension. This is distinct in concept from a unit that is equal to `None`: that indicates that no unit was specified in the data or by the user.

For convenience, there is a unit that is both dimensionless and unscaled: the `dimensionless_unscaled` object:

```
>>> from astropy import units as u
>>> u.dimensionless_unscaled
Unit(dimensionless)
```

Dimensionless quantities are often defined as products or ratios of quantities that are not dimensionless, but whose dimensions cancel out when their powers are multiplied.

Examples

To use the `dimensionless_unscaled` object:

```
>>> u.m / u.m
Unit(dimensionless)
```

For compatibility with the supported unit string formats, this is equivalent to `Unit('')` and `Unit(1)`, though using `u.dimensionless_unscaled` in Python code is preferred for readability:

```
>>> u.dimensionless_unscaled == u.Unit('')
True
>>> u.dimensionless_unscaled == u.Unit(1)
True
```

Note that in many cases, a dimensionless unit may also have a scale. For example:

```
>>> (u.km / u.m).decompose()
Unit(dimensionless with a scale of 1000.0)
>>> (u.km / u.m).decompose() == u.dimensionless_unscaled
False
```

As an example of why you might want to create a scaled dimensionless quantity, say you will be doing many calculations with some big unit-less number, `big_unitless_num = 20000000  # 20 million`, but you want all of your answers to be in multiples of a million. This can be done by dividing `big_unitless_num` by `1e6`, but this requires you to remember that this scaling factor has been applied, which may be difficult to do after many calculations. Instead, create a scaled dimensionless quantity by multiplying a value by `Unit(scale)` to keep track of the scaling factor. For example:

```
>>> scale = 1e6
>>> big_unitless_num = 20 * u.Unit(scale)  # 20 million

>>> some_measurement = 5.0 * u.cm
>>> some_measurement * big_unitless_num
<Quantity 100. 1e+06 cm>
```

To determine if a unit is dimensionless (but regardless of the scale), use the **`physical_type`** property:

```
>>> (u.km / u.m).physical_type
u'dimensionless'
>>> # This also has a scale, so it is not the same as
u.dimensionless_unscaled
>>> (u.km / u.m) == u.dimensionless_unscaled
False
>>> # However, (u.m / u.m) has a scale of 1.0, so it is the same
>>> (u.m / u.m) == u.dimensionless_unscaled
True
```

Enabling Other Units

By default, only the “default” units are searched by **`find_equivalent_units`** and similar methods that do searching. This includes SI, CGS, and astrophysical units. However, you may wish to enable the Imperial or other user-defined units.

Example

To enable Imperial units, do:

```
>>> from astropy.units import imperial
>>> imperial.enable()
>>> u.m.find_equivalent_units()
  Primary name | Unit definition | Aliases
[
  AU           | 1.49598e+11 m   | au, astronomical_unit ,
  Angstrom     | 1e-10 m         | AA, angstrom          ,
  cm           | 0.01 m          | centimeter            ,
  ft           | 0.3048 m        | foot                  ,
  fur          | 201.168 m       | furlong               ,
  inch         | 0.0254 m        |                       ,
  lyr          | 9.46073e+15 m   | lightyear             ,
  m            | irreducible     | meter                 ,
  mi           | 1609.34 m       | mile                  ,
  micron       | 1e-06 m         |                       ,
  mil          | 2.54e-05 m      | thou                  ,
  nmi          | 1852 m          | nauticalmile, NM      ,
  pc           | 3.08568e+16 m   | parsec                ,
  solRad       | 6.957e+08 m     | R_sun, Rsun           ,
  yd           | 0.9144 m        | yard                  ,
]
```

This may also be used with the `with` statement, to temporarily enable additional units:

```
>>> from astropy import units as u
>>> from astropy.units import imperial
>>> with imperial.enable():
...     print(u.m.find_equivalent_units())
      Primary name | Unit definition | Aliases
...
```

To enable only specific units, use **`add_enabled_units`**:

```
>>> from astropy import units as u
>>> from astropy.units import imperial
>>> with u.add_enabled_units([imperial.knot]):
...     print(u.m.find_equivalent_units())
      Primary name | Unit definition | Aliases
...
```

Combining and Defining Units

Units and quantities can be combined together using the regular Python numeric operators.

Example

To combine units and quantities:

```
>>> from astropy import units as u
>>> fluxunit = u.erg / (u.cm ** 2 * u.s)
>>> fluxunit
Unit("erg / (cm2 s)")
>>> 52.0 * fluxunit
<Quantity  52. erg / (cm2 s)>
>>> 52.0 * fluxunit / u.s
<Quantity  52. erg / (cm2 s2)>
```

Units support fractional powers, which retain their precision through complex operations. To do this, it is recommended to use `fractions.Fraction` objects.

Example

To use `fractions.Fraction` objects:

```
>>> from fractions import Fraction
>>> Franklin = u.g ** Fraction(1, 2) * u.cm ** Fraction(3, 2) * u.s
** -1
```

Note

Floating-point powers that are effectively the same as fractions with a denominator less than 10 are implicitly converted to `Fraction` objects under the hood. Therefore, the following are equivalent:

```
>>> x = u.m ** Fraction(1, 3)
>>> x.powers
[Fraction(1, 3)]
>>> x = u.m ** (1. / 3.)
>>> x.powers
[Fraction(1, 3)]
```

Users are free to define new units, either fundamental or compound using the `def_unit` function.

Example

To define new units using the `def_unit` function:

```
>>> bakers_fortnight = u.def_unit('bakers_fortnight', 13 * u.day)
```

The addition of a string gives the new unit a name that will show up when the unit is printed:

```
>>> 10. * bakers_fortnight
<Quantity  10. bakers_fortnight>
```

Creating a new fundamental unit is also possible:

```
>>> titter = u.def_unit('titter')
>>> chuckle = u.def_unit('chuckle', 5 * titter)
>>> laugh = u.def_unit('laugh', 4 * chuckle)
>>> guffaw = u.def_unit('guffaw', 3 * laugh)
>>> rofl = u.def_unit('rofl', 4 * guffaw)
>>> death_by_laughing = u.def_unit('death_by_laughing', 10 * rofl)
>>> (1. * rofl).to(titter)
<Quantity  240. titter>
```

Users can see the definition of a unit and its decomposition via:

```
>>> rofl.represents
Unit("4 guffaw")
>>> rofl.decompose()
Unit("240 titter")
```

By default, custom units are not searched by methods such as `find_equivalent_units`. However, they can be enabled by calling `add_enabled_units`:

```
>>> kmph = u.def_unit('kmph', u.km / u.h)
>>> (u.m / u.s).find_equivalent_units()
There are no equivalent units
>>> u.add_enabled_units([kmph])
<astropy.units.core._UnitContext object at ...>
>>> (u.m / u.s).find_equivalent_units()
  Primary name | Unit definition | Aliases
[
  kmph         | 0.277778 m / s  |         ,
]
```

Decomposing and Composing Units

Reducing a Unit to Its Irreducible Parts

A unit or quantity can be decomposed into its irreducible parts using the `Unit.decompose` or `Quantity.decompose` methods.

Examples

To decompose a unit with `Unit.decompose`:

```
>>> from astropy import units as u
>>> u.Ry
Unit("Ry")
>>> u.Ry.decompose()
Unit("2.17987e-18 kg m2 / s2")
```

You can limit the selection of units that you want to decompose by using the `bases` keyword argument:

```
>>> u.Ry.decompose(bases=[u.m, u.N])
Unit("2.17987e-18 m N")
```

This is also useful to decompose to a particular system. For example, to decompose the Rydberg unit in terms of CGS units:

```
>>> u.Ry.decompose(bases=u.cgs.bases)
Unit("2.17987e-11 cm2 g / s2")
```

Finally, if you want to know how a unit was defined:

```
>>> u.Ry.represents
Unit("13.6057 eV")
```

Automatically Composing a Unit into More Complex Units

Conversely, a unit may be recomposed back into more complex units using the `compose` method. Since there may be multiple equally good results, a list is always returned.

Examples

To recompose a unit with `compose`:

```
>>> x = u.Ry.decompose()
>>> x.compose()
[Unit("Ry"),
 Unit("2.17987e-18 J"),
 Unit("2.17987e-11 erg"),
 Unit("13.6057 eV")]
```

Some other interesting examples:

```
>>> (u.s ** -1).compose()
[Unit("Bq"), Unit("Hz"), Unit("3.7e+10 Ci")]
```

Composition can be combined with Equivalencies:

```
>>> (u.s ** -1).compose(equivalencies=u.spectral())
[Unit("m"),
 Unit("Hz"),
 Unit("J"),
 Unit("Bq"),
 Unit("3.24078e-17 pc"),
 Unit("1.057e-16 lyr"),
 Unit("6.68459e-12 AU"),
 Unit("1.4378e-09 solRad"),
 Unit("0.01 k"),
 Unit("100 cm"),
 Unit("1e+06 micron"),
 Unit("1e+07 erg"),
 Unit("1e+10 Angstrom"),
 Unit("3.7e+10 Ci"),
 Unit("4.58743e+17 Ry"),
 Unit("6.24151e+18 eV")]
```

A name does not exist for every arbitrary derived unit imaginable. In that case, the system will do its best to reduce the unit to the fewest possible symbols:

```
>>> (u.cd * u.sr * u.V * u.s).compose()
[Unit("lm Wb")]
```

Converting Between Systems

Built on top of this functionality is a convenience method to convert between unit systems.

Examples

To convert between unit systems:

```
>>> u.Pa.to_system(u.cgs)
[Unit("10 P / s"), Unit("10 Ba")]
```

There is also a shorthand for this which only returns the first of many possible matches:

```
>>> u.Pa.cgs
Unit("10 P / s")
```

This is equivalent to decomposing into the new system and then composing into the most complex units possible, though **`to_system`** adds some extra logic to return the results sorted in the most useful order:

```
>>> u.Pa.decompose(bases=u.cgs.bases)
Unit("10 g / (cm s2)")
>>> _.compose(units=u.cgs)
[Unit("10 Ba"), Unit("10 P / s")]
```

Magnitudes and Other Logarithmic Units

Magnitudes and logarithmic units such as `dex` and `dB` are used as the logarithm of values relative to some reference value. Quantities with such units are supported in `astropy` via the **`Magnitude`**, **`Dex`**, and **`Decibel`** classes.

Creating Logarithmic Quantities

You can create logarithmic quantities either directly or by multiplication with a logarithmic unit.

Example
To create a logarithmic quantity:

```
>>> import astropy.units as u, astropy.constants as c, numpy as np
>>> u.Magnitude(-10.)
<Magnitude -10. mag>
>>> u.Magnitude(10 * u.ct / u.s)
<Magnitude -2.5 mag(ct / s)>
>>> u.Magnitude(-2.5, "mag(ct/s)")
<Magnitude -2.5 mag(ct / s)>
>>> -2.5 * u.mag(u.ct / u.s)
<Magnitude -2.5 mag(ct / s)>
>>> u.Dex((c.G * u.M_sun / u.R_sun**2).cgs)
<Dex 4.438067627303133 dex(cm / s2)>
>>> np.linspace(2., 5., 7) * u.Unit("dex(cm/s2)")
<Dex [2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ] dex(cm / s2)>
```

Above, we make use of the fact that the units `mag`, `dex`, and `dB` are special in that, when used as functions, they return a **`LogUnit`** instance (**`MagUnit`**, **`DexUnit`**, and **`DecibelUnit`**, respectively). The same happens as required when strings are parsed by **`Unit`**.

As for normal **`Quantity`** objects, you can access the value with the **`value`** attribute. In addition, you can convert to a **`Quantity`** with the physical unit using the **`physical`** attribute:

```
>>> logg = 5. * u.dex(u.cm / u.s**2)
>>> logg.value
5.0
>>> logg.physical
<Quantity 100000. cm / s2>
```

Converting to Different Units

Like **`Quantity`** objects, logarithmic quantities can be converted to different units, be it another logarithmic unit or a physical one.

Example
To convert a logarithmic quantity to a different unit:

```
>>> logg = 5. * u.dex(u.cm / u.s**2)
>>> logg.to(u.m / u.s**2)
<Quantity 1000. m / s2>
>>> logg.to('dex(m/s2)')
<Dex 3. dex(m / s2)>
```

For convenience, the `si` and `cgs` attributes can be used to convert the `Quantity` to base SI or CGS units:

```
>>> logg.si
<Dex 3. dex(m / s2)>
```

Arithmetic and Photometric Applications

Addition and subtraction work as expected for logarithmic quantities, multiplying and dividing the physical units as appropriate. It may be best seen through an example of a photometric reduction.

Example
First, calculate instrumental magnitudes assuming some count rates for three objects:

```
>>> tint = 1000.*u.s
>>> cr_b = ([3000., 100., 15.] * u.ct) / tint
>>> cr_v = ([4000., 90., 25.] * u.ct) / tint
>>> b_i, v_i = u.Magnitude(cr_b), u.Magnitude(cr_v)
>>> b_i, v_i
(<Magnitude [-1.19280314,  2.5       ,  4.55977185] mag(ct / s)>,
 <Magnitude [-1.50514998,  2.61439373,  4.00514998] mag(ct / s)>)
```

Then, the instrumental B-V color is:

```
>>> b_i - v_i
<Magnitude [ 0.31234684, -0.11439373,  0.55462187] mag>
```

Note that the physical unit has become dimensionless. The following step might be used to correct for atmospheric extinction:

```
>>> atm_ext_b, atm_ext_v = 0.12 * u.mag, 0.08 * u.mag
>>> secz = 1./np.cos(45 * u.deg)
>>> b_i0 = b_i - atm_ext_b * secz
>>> v_i0 = v_i - atm_ext_b * secz
>>> b_i0, v_i0
(<Magnitude [-1.36250876,  2.33029437,  4.39006622] mag(ct / s)>,
 <Magnitude [-1.67485561,  2.4446881 ,  3.83544435] mag(ct / s)>)
```

Since the extinction is dimensionless, the units do not change. Now suppose the first star has a known ST magnitude, so we can calculate zero points:

```
>>> b_ref, v_ref = 17.2 * u.STmag, 17.0 * u.STmag
>>> b_ref, v_ref
(<Magnitude 17.2 mag(ST)>, <Magnitude 17. mag(ST)>)
>>> zp_b, zp_v = b_ref - b_i0[0], v_ref - v_i0[0]
>>> zp_b, zp_v
(<Magnitude 18.56250876 mag(s ST / ct)>,
 <Magnitude 18.67485561 mag(s ST / ct)>)
```

Here, `ST` is shorthand for the ST zero-point flux:

```
>>> (0. * u.STmag).to(u.erg/u.s/u.cm**2/u.AA)
<Quantity 3.63078055e-09 erg / (Angstrom cm2 s)>
>>> (-21.1 * u.STmag).to(u.erg/u.s/u.cm**2/u.AA)
<Quantity 1. erg / (Angstrom cm2 s)>
```

Note

At present, only magnitudes defined in terms of luminosity or flux are implemented, since those do not depend on the filter with which the measurement was made. They include absolute and apparent bolometric [M15], ST [H95], and AB [OG83] magnitudes.

Now applying the calibration, we find (note the proper change in units):

```
>>> B, V = b_i0 + zp_b, v_i0 + zp_v
>>> B, V
(<Magnitude [17.2       , 20.89280314, 22.95257499] mag(ST)>,
 <Magnitude [17.        , 21.1195437 , 22.51029996] mag(ST)>)
```

We could convert these magnitudes to another system, for example, ABMag, using appropriate equivalency:

```
>>> V.to(u.ABmag, u.spectral_density(5500.*u.AA))
<Magnitude [16.99023831, 21.10978201, 22.50053827] mag(AB)>
```

This is particularly useful for converting magnitude into flux density. `V` is currently in ST magnitudes, which is based on flux densities per unit wavelength (\(f_\lambda\)). Therefore, we can directly convert `V` into flux density per unit wavelength using the **`to()`** method:

```
>>> flam = V.to(u.erg/u.s/u.cm**2/u.AA)
>>> flam
<Quantity [5.75439937e-16, 1.29473986e-17, 3.59649961e-18] erg /
(Angstrom cm2 s)>
```

To convert `V` to flux density per unit frequency (\(f_\nu\)), we again need the appropriate equivalency, which in this case is the central wavelength of the magnitude band, 5500 Angstroms:

```
>>> lam = 5500 * u.AA
>>> fnu = V.to(u.erg/u.s/u.cm**2/u.Hz, u.spectral_density(lam))
>>> fnu
<Quantity [5.80636959e-27, 1.30643316e-28, 3.62898099e-29] erg / (cm2
Hz s)>
```

We could have used the central frequency instead:

```
>>> nu = 5.45077196e+14 * u.Hz
>>> fnu = V.to(u.erg/u.s/u.cm**2/u.Hz, u.spectral_density(nu))
>>> fnu
<Quantity [5.80636959e-27, 1.30643316e-28, 3.62898099e-29] erg / (cm2
Hz s)>
```

Note

When converting magnitudes to flux densities, the order of operations matters; the value of the unit needs to be established *before* the conversion. For example, `21 * u.ABmag.to(u.erg/u.s/u.cm**2/u.Hz)` will give you 21 times \(f_\nu\) for an AB mag of 1, whereas `(21 * u.ABmag).to(u.erg/u.s/u.cm**2/u.Hz)` will give you \(f_\nu\) for an AB mag of 21.

Suppose we also knew the intrinsic color of the first star, then we can calculate the reddening:

```
>>> B_V0 = -0.2 * u.mag
>>> EB_V = (B - V)[0] - B_V0
>>> R_V = 3.1
>>> A_V = R_V * EB_V
>>> A_B = (R_V+1) * EB_V
>>> EB_V, A_V, A_B
(<Magnitude 0.4 mag>, <Quantity 1.24 mag>, <Quantity 1.64 mag>)
```

Here, you see that the extinctions have been converted to quantities. This happens generally for division and multiplication, since these processes work only for dimensionless magnitudes (otherwise, the physical unit would have to be raised to some power), and **`Quantity`** objects, unlike logarithmic quantities, allow units like `mag / d`.

Note that you can take the automatic unit conversion quite far (perhaps too far, but it is fun). For instance, suppose we also knew the bolometric correction and absolute bolometric magnitude, then we can calculate the distance modulus:

```
>>> BC_V = -0.3 * (u.m_bol - u.STmag)
>>> M_bol = 5.46 * u.M_bol
>>> DM = V[0] - A_V + BC_V - M_bol
>>> BC_V, M_bol, DM
(<Magnitude -0.3 mag(bol / ST)>,
 <Magnitude 5.46 mag(Bol)>,
 <Magnitude 10. mag(bol / Bol)>)
```

With a proper equivalency, we can also convert to distance without remembering the 5-5log rule:

```
>>> radius_and_inverse_area = [(u.pc, u.pc**-2,
...                            lambda x: 1./(4.*np.pi*x**2),
...                            lambda x: np.sqrt(1./(4.*np.pi*x)))]
>>> DM.to(u.pc, equivalencies=radius_and_inverse_area)
<Quantity 1000. pc>
```

NumPy Functions

For logarithmic quantities, most NumPy functions and many array methods do not make sense, hence they are disabled. But you can use those you would expect to work:

```
>>> np.max(v_i)
<Magnitude 4.00514998 mag(ct / s)>
>>> np.std(v_i)
<Magnitude 2.33971149 mag>
```

Note

This is implemented by having a list of supported ufuncs in `units/function/core.py` and by explicitly disabling some array methods in **`FunctionQuantity`**. If you believe a function or method is incorrectly treated, please let us know.

Dimensionless Logarithmic Quantities

Dimensionless quantities are treated somewhat specially in that, if needed, logarithmic quantities will be converted to normal **Quantity** objects with the appropriate unit of `mag`, `dB`, or `dex`. With this, it is possible to use composite units like `mag/d` or `dB/m`, which cannot conveniently be supported as logarithmic units. For instance:

```
>>> dBm = u.dB(u.mW)
>>> signal_in, signal_out = 100. * dBm, 50 * dBm
>>> cable_loss = (signal_in - signal_out) / (100. * u.m)
>>> signal_in, signal_out, cable_loss
(<Decibel 100. dB(mW)>, <Decibel 50. dB(mW)>, <Quantity 0.5 dB / m>)
>>> better_cable_loss = 0.2 * u.dB / u.m
>>> signal_in - better_cable_loss * 100. * u.m
<Decibel 80. dB(mW)>
```

[M15] Mamajek et al., 2015, arXiv:1510.06262

[H95] E.g., Holtzman et al., 1995, PASP 107, 1065

[OG83] Oke, J.B., & Gunn, J. E., 1983, ApJ 266, 713

String Representations of Units

Converting Units to String Representations

You can control the way that **Quantity** and **UnitBase** objects are rendered as strings using the Python Format String Syntax (demonstrated below using f-strings).

For quantities, format specifiers, like `0.003f` will be applied to the **Quantity** value, without affecting the unit. Specifiers like `20s`, which would only apply to a string, will be applied to the whole string representation of the **Quantity**.

Examples

To render **Quantity** or **UnitBase** objects as strings:

```
>>> from astropy import units as u
>>> import numpy as np
>>> q = 10.5 * u.km
>>> q
<Quantity  10.5 km>
>>> f"{q}"
```

```
'10.5 km'
>>> f"{q:+0.03f}"
'+10.500 km'
>>> f"{q:20s}"
'10.5 km                '
```

To format both the value and the unit separately, you can access the **Quantity** class attributes within format strings:

```
>>> q = 10.5 * u.km
>>> q
<Quantity  10.5 km>
>>> f"{q.value:0.003f} in {q.unit:s}"
'10.500 in km'
```

Because `numpy` arrays do not accept most format specifiers, using specifiers like `0.003f` will not work when applied to a `numpy` array or non-scalar **Quantity**. Use **numpy.array_str()** instead. For instance:

```
>>> q = np.linspace(0,1,10) * u.m
>>> "{0} {1}".format(np.array_str(q.value, precision=1), q.unit)
'[0.  0.1 0.2 0.3 0.4 0.6 0.7 0.8 0.9 1. ] m'
```

Examine the NumPy documentation for more examples with **numpy.array_str()**.

Units, or the unit part of a quantity, can also be formatted in a number of different styles. By default, the string format used is referred to as the "generic" format, which is based on syntax of the FITS standard format for representing units, but supports all of the units defined within the **astropy.units** framework, including user-defined units. The format specifier (and **to_string**) functions also take an optional parameter to select a different format, including `"latex"`, `"unicode"`, `"cds"`, and others, defined below.

```
>>> f"{q.value:0.003f} in {q.unit:latex}"
'10.000 in $\\mathrm{km}$'
>>> fluxunit = u.erg / (u.cm ** 2 * u.s)
>>> f"{fluxunit}"
u'erg / (cm2 s)'
>>> print(f"{fluxunit:console}")
 erg
------
s cm^2
>>> f"{fluxunit:latex}"
u'$\\mathrm{\\frac{erg}{s\\,cm^{2}}}$'
>>> f"{fluxunit:>20s}"
```

```
u'       erg / (cm2 s)'
```

The **to_string** method is an alternative way to format units as strings, and is the underlying implementation of the **format**-style usage:

```
>>> fluxunit = u.erg / (u.cm ** 2 * u.s)
>>> fluxunit.to_string('latex')
u'$\\mathrm{\\frac{erg}{s\\,cm^{2}}}$'
```

Creating Units from Strings

Units can also be created from strings in a number of different formats using the **Unit** class:

```
>>> from astropy import units as u
>>> u.Unit("m")
Unit("m")
>>> u.Unit("erg / (s cm2)")
Unit("erg / (cm2 s)")
>>> u.Unit("erg.s-1.cm-2", format="cds")
Unit("erg / (cm2 s)")
```

Note

Creating units from strings requires the use of a specialized parser for the unit language, which results in a performance penalty if units are created using strings. Thus, it is much faster to use unit objects directly (e.g., `unit = u.degree / u.minute`) instead of via string parsing (`unit = u.Unit('deg/min')`). This parser is very useful, however, if your unit definitions are coming from a file format such as FITS or VOTable.

Built-In Formats

astropy.units includes support for parsing and writing the following formats:

- `"fits"`: This is the format defined in the Units section of the FITS Standard. Unlike the “generic” string format, this will only accept or generate units defined in the FITS standard.
- `"vounit"`: The Units in the VO 1.0 standard for representing units in the VO. Again, based on the FITS syntax, but the collection of

supported units is different.

- `"cds"` : Standards for astronomical catalogues from Centre de Données astronomiques de Strasbourg: This is the standard used by Vizier tables, as well as what is used by VOTable versions 1.2 and earlier.
- `"ogip"` : A standard for storing units as recommended by the Office of Guest Investigator Programs (OGIP).

`astropy.units` is also able to write, but not read, units in the following formats:

- `"latex"` : Writes units out using LaTeX math syntax using the IAU Style Manual recommendations for unit presentation. This format is automatically used when printing a unit in the IPython notebook:

  ```
  >>> fluxunit
  ```

  \[\mathrm{\frac{erg}{s\,cm^{2}}}\]
- `"latex_inline"` : Writes units out using LaTeX math syntax using the IAU Style Manual recommendations for unit presentation, using negative powers instead of fractions, as required by some journals (e.g., Apj and AJ). Best suited for unit representation inline with text:

  ```
  >>> fluxunit.to_string('latex_inline')
  ```

  \[\mathrm{erg\,s^{-1}\,cm^{-2}}\]
- `"console"` : Writes a multiline representation of the unit useful for display in a text console:

  ```
  >>> print(fluxunit.to_string('console'))
   erg
  ------
  s cm^2
  ```

- `"unicode"` : Same as `"console"`, except uses Unicode characters:

  ```
  >>> print(u.Ry.decompose().to_string('unicode'))
                  m² kg
  2.1798721×10-¹⁸ ─────
                   s²
  ```

Unrecognized Units

Since many files found in the wild have unit strings that do not correspond to any given standard, **`astropy.units`** also has a consistent way to store and pass around unit strings that did not parse.

Normally, passing an unrecognized unit string raises an exception:

```
>>> # The FITS standard uses 'angstrom', not 'Angstroem'
>>> u.Unit("Angstroem", format="fits")
Traceback (most recent call last):
  ...
ValueError: 'Angstroem' did not parse as fits unit: At col 0, Unit
'Angstroem' not supported by the FITS standard. Did you mean Angstrom
or angstrom? If this is meant to be a custom unit, define it with
'u.def_unit'. To have it recognized inside a file reader or other
code, enable it with 'u.add_enabled_units'. For details, see
https://docs.astropy.org/en/latest/units/combining_and_defining.html
```

However, the **`Unit`** constructor has the keyword argument `parse_strict` that can take one of three values to control this behavior:

- `'raise'` : (default) raise a ValueError exception.
- `'warn'` : emit a Warning, and return an **`UnrecognizedUnit`** instance.
- `'silent'` : return an **`UnrecognizedUnit`** instance.

Examples

To pass an unrecognized unit string:

```
>>> x = u.Unit("Angstroem", format="fits", parse_strict="warn")
WARNING: UnitsWarning: 'Angstroem' did not parse as unit format
'fits': At col 0, 'Angstroem' is not a valid unit in string
'Angstroem' [astropy.units.core]
```

This **`UnrecognizedUnit`** object remembers the original string it was created with, so it can be written back out, but any meaningful operations on it, such as converting to another unit or composing with other units, will fail.

```
>>> x.to_string()
'Angstroem'
>>> x.to(u.km)
Traceback (most recent call last):
  ...
ValueError: The unit 'Angstroem' is unrecognized.  It can not be
converted to other units.
```

```
>>> x / u.m
Traceback (most recent call last):
  ...
ValueError: The unit 'Angstroem' is unrecognized, so all arithmetic
operations with it are invalid.
```

Equivalencies

The unit module has machinery for supporting equivalences between different units in certain contexts, namely when equations can uniquely relate a value in one unit to a different unit. A good example is the equivalence between wavelength, frequency, and energy for specifying a wavelength of radiation. Normally these units are not convertible, but when understood as representing light, they are convertible in certain contexts. Here we describe how to use the equivalencies included in **`astropy.units`** and how to define new equivalencies.

Equivalencies are used by passing a list of equivalency pairs to the `equivalencies` keyword argument of **`Quantity.to`** or **`Unit.to`** methods. Alternatively, if a larger piece of code needs the same equivalencies, you can set them for a given context.

Built-In Equivalencies

How to Convert Parallax to Distance

The length unit *parsec* is defined such that a star one parsec away will exhibit a 1-arcsecond parallax. (Think of it as a contraction between *parallax* and *arcsecond*.)

The **`parallax()`** function handles conversions between parallax angles and length.

In general, you should not be able to change units of length into angles or vice versa, so **`to()`** raises an exception:

```
>>> from astropy import units as u
>>> (0.8 * u.arcsec).to(u.parsec)
Traceback (most recent call last):
  ...
UnitConversionError: 'arcsec' (angle) and 'pc' (length) are not
convertible
```

To trigger the conversion between parallax angle and distance, provide **`parallax()`** as the optional keyword argument (`equivalencies=`) to the to() method.

```
>>> (0.8 * u.arcsec).to(u.parsec, equivalencies=u.parallax())
<Quantity 1.25 pc>
```

Angles as Dimensionless Units

Angles are treated as a physically distinct type, which usually helps to avoid mistakes. However, this is not very handy when working with units related to rotational energy or the small angle approximation. (Indeed, this double-sidedness underlies why radians went from a supplementary to derived unit.) The function dimensionless_angles() provides the required equivalency list that helps convert between angles and dimensionless units. It is somewhat different from all others in that it allows an arbitrary change in the number of powers to which radians is raised (i.e., including zero and thus dimensionless).

Examples

Normally the following would raise exceptions:

```
>>> from astropy import units as u
>>> u.degree.to('')
Traceback (most recent call last):
  ...
UnitConversionError: 'deg' (angle) and '' (dimensionless) are not
convertible
>>> (u.kg * u.m**2 * (u.cycle / u.s)**2).to(u.J)
Traceback (most recent call last):
  ...
UnitConversionError: 'cycle2 kg m2 / s2' and 'J' (energy) are not
convertible
```

But when passing the proper conversion function, dimensionless_angles(), it works.

```
>>> u.deg.to('', equivalencies=u.dimensionless_angles())
0.017453292519943295
>>> (0.5e38 * u.kg * u.m**2 * (u.cycle / u.s)**2).to(u.J,
...
equivalencies=u.dimensionless_angles())
<Quantity 1.9739208802178715e+39 J>
>>> import numpy as np
>>> np.exp((1j*0.125*u.cycle).to('',
equivalencies=u.dimensionless_angles()))
<Quantity  0.70710678+0.70710678j>
```

In an example with complex numbers you may well be doing a fair number of similar calculations. For such situations, there is the option to set default equivalencies.

In some situations, this equivalency may behave differently than anticipated. For instance, it might at first seem reasonable to use it for converting from an angular velocity \(\omega\) in radians per second to the corresponding frequency \(f\) in hertz (i.e., to implement \(f=\omega/2\pi\)). However, attempting this yields:

```
>>> (1*u.rad/u.s).to(u.Hz, equivalencies=u.dimensionless_angles())
<Quantity 1. Hz>
>>> (1*u.cycle/u.s).to(u.Hz, equivalencies=u.dimensionless_angles())
<Quantity 6.283185307179586 Hz>
```

Here, we might have expected ~0.159 Hz in the first example and 1 Hz in the second. However, `dimensionless_angles()` converts to radians per second and then drops radians as a unit. The implicit mistake made in these examples is that the unit Hz is taken to be equivalent to cycles per second, which it is not (it is just “per second”). This realization also leads to the solution: to use an explicit equivalency between cycles per second and hertz:

```
>>> (1*u.rad/u.s).to(u.Hz, equivalencies=[(u.cy/u.s, u.Hz)])
<Quantity 0.15915494309189535 Hz>
>>> (1*u.cy/u.s).to(u.Hz, equivalencies=[(u.cy/u.s, u.Hz)])
<Quantity 1. Hz>
```

Spectral Units

`spectral()` is a function that returns an equivalency list to handle conversions between wavelength, frequency, energy, and wave number.

As mentioned with parallax units, we pass a list of equivalencies (in this case, the result of `spectral()`) as the second argument to the `to()` method and wavelength, and then frequency and energy can be converted.

```
>>> ([1000, 2000] * u.nm).to(u.Hz, equivalencies=u.spectral())
<Quantity [2.99792458e+14, 1.49896229e+14] Hz>
>>> ([1000, 2000] * u.nm).to(u.eV, equivalencies=u.spectral())
<Quantity [1.23984193, 0.61992096] eV>
```

These equivalencies even work with non-base units:

```
>>> # Inches to calories
>>> from astropy.units import imperial
>>> imperial.inch.to(imperial.Cal, equivalencies=u.spectral())
1.869180759162485e-27
```

Spectral (Doppler) Equivalencies

Spectral equivalencies allow you to convert between wavelength, frequency, energy, and wave number, but not to velocity, which is frequently the quantity of interest.

It is fairly convenient to define the equivalency, but note that there are different conventions. In these conventions f_0 is the rest frequency, f is the observed frequency, V is the velocity, and c is the speed of light:

- Radio $V = c \frac{f_0 - f}{f_0} ; f(V) = f_0 (1 - V/c)$
- Optical $V = c \frac{f_0 - f}{f } ; f(V) = f_0 (1 + V/c)^{-1}$
- Relativistic $V = c \frac{f_0^2 - f^2}{f_0^2 + f^2} ; f(V) = f_0 \frac{\left(1 - (V/c)^2\right)^{1/2}}{(1+V/c)}$

These three conventions are implemented in **`astropy.units.equivalencies`** as **`doppler_optical()`**, **`doppler_radio()`**, and **`doppler_relativistic()`**.

Example

To define an equivalency:

```
>>> restfreq = 115.27120 * u.GHz  # rest frequency of 12 CO 1-0 in
GHz
>>> freq_to_vel = u.doppler_radio(restfreq)
>>> (116e9 * u.Hz).to(u.km / u.s, equivalencies=freq_to_vel)
<Quantity -1895.4321928669085 km / s>
```

Spectral Flux and Luminosity Density Units

There is also support for spectral flux and luminosity density units, their equivalent surface brightness units, and integrated flux units. Their use is more complex, since it is necessary to also supply the location in the spectrum for which the conversions will be done, and the units of those spectral locations. The function that handles these unit conversions is **`spectral_density()`**. This function takes as its arguments the **`Quantity`** for the spectral location.

Example

To perform unit conversions with **`spectral_density()`**:

```
>>> (1.5 * u.Jy).to(u.photon / u.cm**2 / u.s / u.Hz,
...                     equivalencies=u.spectral_density(3500 * u.AA))
<Quantity 2.6429114293019694e-12 ph / (cm2 Hz s)>
>>> (1.5 * u.Jy).to(u.photon / u.cm**2 / u.s / u.micron,
...                     equivalencies=u.spectral_density(3500 * u.AA))
<Quantity 6467.9584789120845 ph / (cm2 micron s)>
>>> a = 1. * (u.photon / u.s / u.angstrom)
>>> a.to(u.erg / u.s / u.Hz,
...      equivalencies=u.spectral_density(5500 * u.AA))
```

```
<Quantity 3.6443382634999996e-23 erg / (Hz s)>
>>> w = 5000 * u.AA
>>> a = 1. * (u.erg / u.cm**2 / u.s)
>>> b = a.to(u.photon / u.cm**2 / u.s, u.spectral_density(w))
>>> b
<Quantity 2.51705828e+11 ph / (cm2 s)>
>>> b.to(a.unit, u.spectral_density(w))
<Quantity 1. erg / (cm2 s)>
```

Brightness Temperature and Surface Brightness Equivalency

There is an equivalency between surface brightness (flux density per area) and brightness temperature. This equivalency is often referred to as “Antenna Gain” since, at a given frequency, telescope brightness sensitivity is unrelated to aperture size, but flux density sensitivity is, so this equivalency is only dependent on the aperture size. See Tools of Radio Astronomy for details.

Note

The brightness temperature mentioned here is the Rayleigh-Jeans equivalent temperature, which results in a linear relation between flux and temperature. This is the convention that is most often used in relation to observations, but if you are interested in computing the *exact* temperature of a blackbody function that would produce a given flux, you should not use this equivalency.

Examples

The **`brightness_temperature`** equivalency requires the beam area and frequency as arguments. Recalling that the area of a 2D Gaussian is \(2 \pi \sigma^2\) (see wikipedia), here is an example:

```
>>> import numpy as np
>>> beam_sigma = 50*u.arcsec
>>> omega_B = 2 * np.pi * beam_sigma**2
>>> freq = 5 * u.GHz
>>> (1*u.Jy/omega_B).to(u.K,
equivalencies=u.brightness_temperature(freq))
<Quantity 3.526295144567176 K>
```

If you have beam full-width half-maxima (FWHM), which are often quoted and are the values stored in the FITS header keywords BMAJ and BMIN, a more appropriate example converts the FWHM to sigma:

```
>>> import numpy as np
>>> beam_fwhm = 50*u.arcsec
>>> fwhm_to_sigma = 1. / (8 * np.log(2))**0.5
>>> beam_sigma = beam_fwhm * fwhm_to_sigma
>>> omega_B = 2 * np.pi * beam_sigma**2
>>> freq = 5 * u.GHz
>>> (1*u.Jy/omega_B).to(u.K,
equivalencies=u.brightness_temperature(freq))
<Quantity 19.553932298231704 K>
```

You can also convert between `Jy/beam` and `K` by specifying the beam area:

```
>>> import numpy as np
>>> beam_fwhm = 50*u.arcsec
>>> fwhm_to_sigma = 1. / (8 * np.log(2))**0.5
>>> beam_sigma = beam_fwhm * fwhm_to_sigma
>>> omega_B = 2 * np.pi * beam_sigma**2
>>> freq = 5 * u.GHz
>>> (1*u.Jy/u.beam).to(u.K, u.brightness_temperature(freq,
beam_area=omega_B))
<Quantity 19.553932298231704 K>
```

Beam Equivalency

Radio data, especially from interferometers, is often produced in units of `Jy/beam`. Converting this number to a beam-independent value (e.g., `Jy/sr`), can be done with the **`beam_angular_area`** equivalency.

Example

To convert units of `Jy/beam` to `Jy/sr`:

```
>>> import numpy as np
>>> beam_fwhm = 50*u.arcsec
>>> fwhm_to_sigma = 1. / (8 * np.log(2))**0.5
>>> beam_sigma = beam_fwhm * fwhm_to_sigma
>>> omega_B = 2 * np.pi * beam_sigma**2
>>> (1*u.Jy/u.beam).to(u.MJy/u.sr,
equivalencies=u.beam_angular_area(omega_B))
<Quantity 15.019166691021288 MJy / sr>
```

Note that the radio_beam package deals with beam input/output and various operations more directly.

Temperature Energy Equivalency

This equivalency allows conversion between temperature and its equivalent in energy (i.e., the temperature multiplied by the Boltzmann constant), usually expressed in electronvolts. This is used frequently for observations at high-energy, be it for solar or X-ray astronomy.

Example

To convert between temperature and its equivalent in energy:

```
>>> import astropy.units as u
>>> t_k = 1e6 * u.K
>>> t_k.to(u.eV, equivalencies=u.temperature_energy())
<Quantity 86.17332384960955 eV>
```

Thermodynamic Temperature Equivalency

This `thermodynamic_temperature()` equivalency allows conversion between `Jy/beam` and “thermodynamic temperature”, \(T_{CMB}\), in Kelvins.

Examples

To convert between `Jy/beam` and thermodynamic temperature:

```
>>> import astropy.units as u
>>> nu = 143 * u.GHz
>>> t_k = 0.002632051878 * u.K
>>> t_k.to(u.MJy / u.sr,
equivalencies=u.thermodynamic_temperature(nu))
<Quantity 1. MJy / sr>
```

By default, this will use the \(T_{CMB}\) value for the default cosmology in `astropy`, but it is possible to specify a custom \(T_{CMB}\) value for a specific cosmology as the second argument to the equivalency:

```
>>> from astropy.cosmology import WMAP9
>>> t_k.to(u.MJy / u.sr,
equivalencies=u.thermodynamic_temperature(nu, T_cmb=WMAP9.Tcmb0))
<Quantity 0.99982392 MJy / sr>
```

Molar Mass AMU Equivalency

This equivalency allows conversion between the atomic mass unit and the equivalent g/mol. For context, refer to the NIST definition of SI Base Units.

Example

To convert between atomic mass unit and the equivalent g/mol:

```
>>> import astropy.units as u
>>> import astropy.constants as const
>>> x = 1 * (u.g / u.mol)
>>> y = 1 * u.u
>>> x.to(u.u, equivalencies=u.molar_mass_amu())
<Quantity 1.0 u>
>>> y.to(u.g/u.mol, equivalencies=u.molar_mass_amu())
<Quantity 1.0 g / mol>
```

Pixel and Plate Scale Equivalencies

These equivalencies are for converting between angular scales and either linear scales in the focal plane or distances in units of the number of pixels.

Examples

Suppose you are working with cutouts from the Sloan Digital Sky Survey, which defaults to a pixel scale of 0.4 arcseconds per pixel, and want to know the true size of something that you measure to be 240 pixels across in the cutout image:

```
>>> import astropy.units as u
>>> sdss_pixelscale = u.pixel_scale(0.4*u.arcsec/u.pixel)
>>> (240*u.pixel).to(u.arcmin, sdss_pixelscale)
<Quantity 1.6 arcmin>
```

Or maybe you are designing an instrument for a telescope that someone told you has an inverse plate scale of 7.8 meters per radian (for your desired focus), and you want to know how big your pixels need to be to cover half an arcsecond:

```
>>> import astropy.units as u
>>> tel_platescale = u.plate_scale(7.8*u.m/u.radian)
>>> (0.5*u.arcsec).to(u.micron, tel_platescale)
<Quantity 18.9077335632719 micron>
```

The pixel scale equivalency can also work in more general context, where the scale is specified as any quantity that is reducible to `<composite unit>/u.pix` or `u.pix/<composite unit>` (that is, the dimensionality of `u.pix` is 1 or -1). For instance, you may define the dots per inch (DPI) for a digital image to calculate its physical size:

```
>>> import astropy.units as u
>>> dpi = u.pixel_scale(100 * u.pix / u.imperial.inch)
>>> (1024 * u.pix).to(u.cm, dpi)
<Quantity 26.0096 cm>
```

Photometric Zero Point Equivalency

This equivalency provides a way to move between photometric systems (i.e., those defined relative to a particular zero-point flux) and absolute fluxes. This is most useful in conjunction with support for Magnitudes and Other Logarithmic Units.

Example

Suppose you are observing a target with a filter with a reported standard zero point of 3631.1 Jy:

```
>>> target_flux = 1.2 * u.nanomaggy
>>> zero_point_star_equiv = u.zero_point_flux(3631.1 * u.Jy)
>>> u.Magnitude(target_flux.to(u.AB, zero_point_star_equiv))
<Magnitude 22.30195136 mag(AB)>
```

Reduced Hubble Constant and “little-h” Equivalency

The dimensionless version of the Hubble constant — often known as “little h” — is a frequently used quantity in extragalactic astrophysics. It is also widely known as the bane of beginners’ existence in such fields (See e.g., the title of this paper, which also provides valuable advice on the use of little h). `astropy` provides an equivalency that helps keep this straight in at least some of these cases, by providing a way to convert to/from physical to “little h” units.

Examples

To convert to or from physical to “little h” units:

```
>>> import astropy.units as u
>>> H0_70 = 70 * u.km/u.s / u.Mpc
>>> distance = 70 * (u.Mpc/u.littleh)
>>> distance.to(u.Mpc, u.with_H0(H0_70))
<Quantity 100.0 Mpc>
>>> luminosity = 0.49 * u.Lsun * u.littleh**-2
>>> luminosity.to(u.Lsun, u.with_H0(H0_70))
<Quantity 1.0 solLum>
```

Note the unit name `littleh`: while this unit is usually expressed in the literature as just `h`, here it is `littleh` to avoid confusion with “hours.”

If no argument is given (or the argument is **None**), this equivalency assumes the `H0` from the current default cosmology:

```
>>> distance = 100 * (u.Mpc/u.littleh)
>>> distance.to(u.Mpc, u.with_H0())
<Quantity 147.79781259 Mpc>
```

This equivalency also allows a common magnitude formulation of little h scaling:

```
>>> mag_quantity = 12 * (u.mag - u.MagUnit(u.littleh**2))
>>> mag_quantity
<Magnitude 12. mag(1 / littleh2)>
>>> mag_quantity.to(u.mag, u.with_H0(H0_70))
<Quantity 11.2254902 mag>
```

Temperature Equivalency

The `temperature()` equivalency allows conversion between the Celsius, Fahrenheit, Rankine and Kelvin.

Example

To convert between temperature scales:

```
>>> import astropy.units as u
>>> temp_C = 0 * u.Celsius
>>> temp_Kelvin = temp_C.to(u.K, equivalencies=u.temperature())
>>> temp_Kelvin
<Quantity 273.15 K>
>>> temp_F = temp_C.to(u.imperial.deg_F,
equivalencies=u.temperature())
>>> temp_F
<Quantity 32. deg_F>
>>> temp_R = temp_C.to(u.imperial.deg_R,
equivalencies=u.temperature())
>>> temp_R
<Quantity 491.67 deg_R>
```

Note

You can also use `u.deg_C` instead of `u.Celsius`.

Mass-Energy Equivalency

In a special relativity context, mass and energy can be equivalent units. For instance:

```
>>> import astropy.units as u
>>> (1 * u.g).to(u.eV, u.mass_energy())
<Quantity 5.60958865e+32 eV>
```

Writing New Equivalencies

An equivalence list is a list of tuples, where each tuple has four elements:

```
(from_unit, to_unit, forward, backward)
```

`from_unit` and `to_unit` are the equivalent units. `forward` and `backward` are functions that convert values between those units. `forward` and `backward` are optional, and if omitted such an equivalency declares that the two units should be taken as equivalent. The functions must take and return non-Quantity objects to avoid infinite recursion; See A More Complex Example:

Spectral Doppler Equivalencies for more details.

Examples

Until 1964, the metric liter was defined as the volume of 1kg of water at 4°C at 760mm mercury pressure. Volumes and masses are not normally directly convertible, but if we hold the constants in the 1964 definition of the liter as true, we could build an equivalency for them:

```
>>> liters_water = [
...    (u.l, u.g, lambda x: 1000.0 * x, lambda x: x / 1000.0)
... ]
>>> u.l.to(u.kg, 1, equivalencies=liters_water)
1.0
```

Note that the equivalency can be used with any other compatible unit:

```
>>> from astropy.units import imperial
>>> imperial.gallon.to(imperial.pound, 1, equivalencies=liters_water)
8.345404463333525
```

And it also works in the other direction:

```
>>> imperial.lb.to(imperial.pint, 1, equivalencies=liters_water)
0.9586114172355459
```

A More Complex Example: Spectral Doppler Equivalencies

We show how to define an equivalency using the radio convention for CO 1-0. This function is already defined in `doppler_radio()`, but this example is illustrative:

```
>>> from astropy.constants import si
>>> restfreq = 115.27120  # rest frequency of 12 CO 1-0 in GHz
>>> freq_to_vel = [(u.GHz, u.km/u.s,
... lambda x: (restfreq-x) / restfreq * si.c.to_value('km/s'),
... lambda x: (1-x/si.c.to_value('km/s')) * restfreq )]
>>> u.Hz.to(u.km / u.s, 116e9, equivalencies=freq_to_vel)
-1895.4321928669262
>>> (116e9 * u.Hz).to(u.km / u.s, equivalencies=freq_to_vel)
<Quantity -1895.4321928669262 km / s>
```

Note that once this is defined for GHz and km/s, it will work for all other units of frequency and velocity. `x` is converted from the input frequency unit (e.g., Hz) to GHz before being passed to `lambda x:`. Similarly, the return value is assumed to be in units of `km/s`, which is why the `.value` of `c` is used instead of the constant.

Displaying Available Equivalencies

The `find_equivalent_units()` method also understands equivalencies.

Example

Without passing equivalencies, there are three compatible units for `Hz` in the standard set:

```
>>> u.Hz.find_equivalent_units()
  Primary name | Unit definition | Aliases
[
  Bq           | 1 / s           | becquerel    ,
  Ci           | 3.7e+10 / s     | curie        ,
  Hz           | 1 / s           | Hertz, hertz ,
]
```

However, when passing the spectral equivalency, you can see there are all kinds of things that `Hz` can be converted to:

```
>>> u.Hz.find_equivalent_units(equivalencies=u.spectral())
  Primary name | Unit definition        | Aliases
[
  AU           | 1.49598e+11 m          | au, astronomical_unit ,
  Angstrom     | 1e-10 m                | AA, angstrom          ,
  Bq           | 1 / s                  | becquerel             ,
  Ci           | 3.7e+10 / s            | curie                 ,
  Hz           | 1 / s                  | Hertz, hertz          ,
  J            | kg m2 / s2             | Joule, joule          ,
  Ry           | 2.17987e-18 kg m2 / s2 | rydberg               ,
  cm           | 0.01 m                 | centimeter            ,
  eV           | 1.60218e-19 kg m2 / s2 | electronvolt          ,
  earthRad     | 6.3781e+06 m           | R_earth, Rearth       ,
  erg          | 1e-07 kg m2 / s2       |                       ,
  jupiterRad   | 7.1492e+07 m           | R_jup, Rjup, R_jupiter,
Rjupiter ,
  k            | 100 / m                | Kayser, kayser        ,
  lyr          | 9.46073e+15 m          | lightyear             ,
  m            | irreducible            | meter                 ,
  micron       | 1e-06 m                |                       ,
  pc           | 3.08568e+16 m          | parsec                ,
  solRad       | 6.957e+08 m            | R_sun, Rsun           ,
]
```

Using Equivalencies in Larger Pieces of Code

Sometimes you may have an involved calculation where you are regularly switching back and forth between equivalent units. For these cases, you can set equivalencies that will by default be used, in a way similar to how you can enable other units.

Examples

To enable radians to be treated as a dimensionless unit:

```
>>> import astropy.units as u
>>> u.set_enabled_equivalencies(u.dimensionless_angles())
<astropy.units.core._UnitContext object at ...>
>>> u.deg.to('')
0.017453292519943295
```

Here, any list of equivalencies could be used, or you could add, for example, `spectral()` and `spectral_density()` (since these return lists, they should indeed be combined by adding them together).

The disadvantage of the above approach is that you may forget to turn the default off (done by giving an empty argument). To automate this, a context manager is provided:

```
>>> import astropy.units as u
>>> with u.set_enabled_equivalencies(u.dimensionless_angles()):
...     phase = 0.5 * u.cycle
...     c = np.exp(1j*phase)
>>> c
<Quantity (-1+1.2246063538223773e-16j) >
```

Using Prior Versions of Constants

By default, `astropy.units` are initialized upon first import to use the current versions of `astropy.constants`. For units to initialize properly to a prior version of constants, the constants versions must be set before the first import of `astropy.units` or `astropy.constants`.

This is accomplished using ScienceState classes in the top-level package. Setting the prior versions at the start of a Python session will allow consistent units.

Example

To initialize units to a prior version of constants:

```
>>> import astropy
>>> astropy.physical_constants.set('codata2010')
<ScienceState physical_constants: 'codata2010'>
>>> astropy.astronomical_constants.set('iau2012')
<ScienceState astronomical_constants: 'iau2012'>
>>> import astropy.units as u
>>> import astropy.constants as const
>>> (const.M_sun / u.M_sun).to(u.dimensionless_unscaled) - 1
<Quantity 0.>
>>> const.M_sun
  Name   = Solar mass
  Value  = 1.9891e+30
  Uncertainty  = 5e+25
  Unit  = kg
  Reference = Allen's Astrophysical Quantities 4th Ed.
```

If `astropy.units` has already been imported, a RuntimeError is raised.

Low-Level Unit Conversion

Conversion of quantities from one unit to another is handled using the `Quantity.to` method. This page describes some low-level features for handling unit conversion that are rarely required in user code.

There are two ways of handling conversions between units.

Direct Conversion

In this case, given a source and destination unit, the values in the new units are returned.

```
>>> from astropy import units as u
>>> u.pc.to(u.m, 3.26)
1.0059308915661856e+17
```

This converts 3.26 parsecs to meters.

Arrays are permitted as arguments.

```
>>> u.h.to(u.s, [1, 2, 5, 10.1])
array([  3600.,   7200.,  18000.,  36360.])
```

Incompatible Conversions

If you attempt to convert to a incompatible unit, an exception will result:

```
>>> cms = u.cm / u.s
>>> cms.to(u.km)
Traceback (most recent call last):
  ...
UnitConversionError: 'cm / s' (speed) and 'km' (length) are not
convertible
```

You can check whether a particular conversion is possible using the **`is_equivalent`** method:

```
>>> u.m.is_equivalent(u.pc)
True
>>> u.m.is_equivalent("second")
False
>>> (u.m ** 3).is_equivalent(u.l)
True
```

Acknowledgments

This code was originally based on the pynbody units module written by Andrew Pontzen, who has granted the Astropy Project permission to use the code under a BSD license.

See Also

- FITS Standard for units in FITS.
- The Units in the VO 1.0 Standard for representing units in the VO.
- OGIP Units: A standard for storing units in OGIP FITS files.
- Standards for astronomical catalogues units.
- IAU Style Manual.
- A table of astronomical unit equivalencies.

Performance Tips

If you are attaching units to arrays to make **`Quantity`** objects, multiplying arrays by units will result in the array being copied in memory, which will slow things down. Furthermore, if you are multiplying an array by a composite unit, the array will be copied for each individual multiplication. Thus, in the following case, the array is copied four successive times:

```
In [1]: array = np.random.random(10000000)

In [2]: %timeit array * u.m / u.s / u.kg / u.sr
92.5 ms ± 2.52 ms per loop (mean ± std. dev. of 7 runs, 10 loops
each)
```

There are several ways to speed this up. First, when you are using composite units, ensure that the entire unit gets evaluated first, then attached to the array. You can do this by using parentheses as for any other operation:

```
In [3]: %timeit array * (u.m / u.s / u.kg / u.sr)
21.5 ms ± 886 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In this case, this has sped things up by a factor of 4x. If you use a composite unit several times in your code, another approach is to create a constant at the top of your code for this unit and use it subsequently:

```
In [4]: UNIT_MSKGSR = u.m / u.s / u.kg / u.sr

In [5]: %timeit array * UNIT_MSKGSR
22.2 ms ± 551 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In this case and the case with brackets, the array is still copied once when creating the `Quantity`. If you want to avoid any copies altogether, you can make use of the `<<` operator to attach the unit to the array:

```
In [6]: %timeit array << u.m / u.s / u.kg / u.sr
47.1 µs ± 5.77 µs per loop (mean ± std. dev. of 7 runs, 10000 loops
each)
```

Note that these are now **microseconds**, so this is 2000x faster than the original case with no brackets. Note that brackets are not needed when using `<<` since `*` and `/` have a higher precedence, so the unit will be evaluated first. When using `<<`, be aware that because the data is not being copied, changing the original array will also change the `Quantity` object.

Note that for composite units, you will definitely see an impact if you can pre-compute the composite unit:

```
In [7]: %timeit array << UNIT_MSKGSR
6.51 µs ± 112 ns per loop (mean ± std. dev. of 7 runs, 100000 loops
each)
```

Which is over 10000x faster than the original example. See Creating and Converting Quantities without Copies for more details about the `<<` operator.

Reference/API

astropy.units.quantity Module

This module defines the **Quantity** object, which represents a number with some associated units. **Quantity** objects support operations like ordinary numbers, but will deal with unit conversions internally.

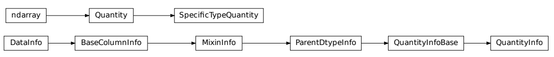
Functions

allclose(a, b[, rtol, atol, equal_nan])	Whether two arrays are element-wise equal within a tolerance.
isclose(a, b[, rtol, atol, equal_nan])	Return a boolean array where two arrays are element-wise equal within a tolerance.

Classes

Quantity(value[, unit, dtype, copy, order, …])	A **Quantity** represents a number with some associated unit.
SpecificTypeQuantity(value[, unit, dtype, …])	Superclass for Quantities of specific physical type.
QuantityInfoBase([bound])	
QuantityInfo([bound])	Container for meta information like name, description, format.

Class Inheritance Diagram



astropy.units Package

This subpackage contains classes and functions for defining and converting between different physical units.

This code is adapted from the pynbody units module written by Andrew Pontzen, who has granted the Astropy project permission to use the code under a BSD license.

Functions

`add_enabled_equivalencies`(equivalencies)	Adds to the equivalencies enabled in the unit registry.
`add_enabled_units`(units)	Adds to the set of units enabled in the unit registry.
`allclose`(a, b[, rtol, atol, equal_nan])	Whether two arrays are element-wise equal within a tolerance.
`beam_angular_area`(beam_area)	Convert between the `beam` unit, which is commonly used to express the area of a radio telescope resolution element, and an area on the sky.
`brightness_temperature`(frequency[, beam_area])	Defines the conversion between Jy/sr and "brightness temperature", \(T_B\), in Kelvins.
`def_physical_type`(unit, name)	Adds a new physical unit mapping.
`def_unit`(s[, represents, doc, format, …])	Factory function for defining new units.
`dimensionless_angles`()	Allow angles to be equivalent to dimensionless (with 1 rad = 1 m/m = 1).
`doppler_optical`(rest)	Return the equivalency pairs for the optical convention for velocity.
`doppler_radio`(rest)	Return the equivalency pairs for the radio convention for velocity.
`doppler_relativistic`(rest)	Return the equivalency pairs for the relativistic convention for velocity.
`get_current_unit_registry`()	
`get_physical_type`(unit)	Given a unit, returns the name of the physical quantity it represents.
`isclose`(a, b[, rtol, atol, equal_nan])	Return a boolean array where two arrays are element-wise equal within a tolerance.
`logarithmic`()	Allow logarithmic units to be converted to dimensionless fractions
`mass_energy`()	Returns a list of equivalence pairs that handle the conversion between mass and energy.
`molar_mass_amu`()	Returns the equivalence between amu and molar mass.
`parallax`()	Returns a list of equivalence pairs that handle the conversion between parallax angle and distance.
`pixel_scale`(pixscale)	Convert between pixel distances (in units of `pix`) and other units, given a particular `pixscale`.
`plate_scale`(platescale)	Convert between lengths (to be interpreted as lengths in the focal plane) and angular units with a specified `platescale`.
`quantity_input`([func])	A decorator for validating the units of arguments to functions.
`set_enabled_equivalencies`(equivalencies)	Sets the equivalencies enabled in the unit registry.
`set_enabled_units`(units)	Sets the units enabled in the unit registry.
`spectral`()	Returns a list of equivalence pairs that handle spectral wavelength, wave number, frequency, and energy equivalences.

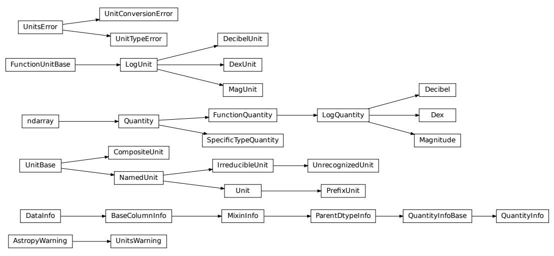
`spectral_density`(wav[, factor])	Returns a list of equivalence pairs that handle spectral density with regard to wavelength and frequency.
`temperature`()	Convert between Kelvin, Celsius, Rankine and Fahrenheit here because Unit and CompositeUnit cannot do addition or subtraction properly.
`temperature_energy`()	Convert between Kelvin and keV(eV) to an equivalent amount.
`thermodynamic_temperature`(frequency[, T_cmb])	Defines the conversion between Jy/sr and “thermodynamic temperature”, \(T_{CMB}\), in Kelvins.
`with_H0`([H0])	Convert between quantities with little-h and the equivalent physical units.
`zero_point_flux`(flux0)	An equivalency for converting linear flux units (“maggys”) defined relative to a standard source into a standardized system.

Classes

`CompositeUnit`(scale, bases, powers[, …])	Create a composite unit using expressions of previously defined units.
`Decibel`(value[, unit, dtype, copy, order, …])	
`DecibelUnit`([physical_unit, function_unit])	Logarithmic physical units expressed in dB
`Dex`(value[, unit, dtype, copy, order, …])	
`DexUnit`([physical_unit, function_unit])	Logarithmic physical units expressed in magnitudes
`FunctionQuantity`(value[, unit, dtype, copy, …])	A representation of a (scaled) function of a number with a unit.
`FunctionUnitBase`([physical_unit, function_unit])	Abstract base class for function units.
`IrreducibleUnit`(st[, doc, format, namespace])	Irreducible units are the units that all other units are defined in terms of.
`LogQuantity`(value[, unit, dtype, copy, …])	A representation of a (scaled) logarithm of a number with a unit
`LogUnit`([physical_unit, function_unit])	Logarithmic unit containing a physical one
`MagUnit`([physical_unit, function_unit])	Logarithmic physical units expressed in magnitudes
`Magnitude`(value[, unit, dtype, copy, order, …])	
`NamedUnit`(st[, doc, format, namespace])	The base class of units that have a name.
`PrefixUnit`(s[, represents, format, …])	A unit that is simply a SI-prefixed version of another unit.
`Quantity`(value[, unit, dtype, copy, order, …])	A `Quantity` represents a number with some associated unit.
`QuantityInfo`([bound])	Container for meta information like name, description, format.
`QuantityInfoBase`([bound])	
`SpecificTypeQuantity`(value[, unit, dtype, …])	Superclass for Quantities of specific physical type.

Unit(s[, represents, format, namespace, …])	The main unit class.
UnitBase()	Abstract base class for units.
UnitConversionError	Used specifically for errors related to converting between units or interpreting units in terms of other units.
UnitTypeError	Used specifically for errors in setting to units not allowed by a class.
UnitsError	The base class for unit-specific exceptions.
UnitsWarning	The base class for unit-specific warnings.
UnrecognizedUnit(st[, doc, format, namespace])	A unit that did not parse correctly.

Class Inheritance Diagram



astropy.units.format Package

A collection of different unit formats.

Functions

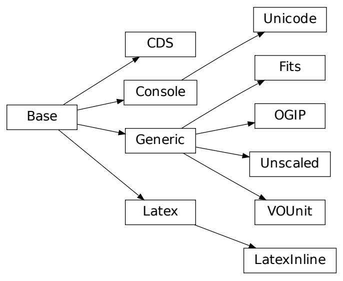
get_format([format])	Get a formatter by name.

Classes

Base(*args, **kwargs)	The abstract base class of all unit formats.
Generic(*args, **kwargs)	A “generic” format.
CDS(*args, **kwargs)	Support the Centre de Données astronomiques de Strasbourg Standards for Astronomical Catalogues 2.0 format, and the complete set of supported units.
Console(*args, **kwargs)	Output-only format for to display pretty formatting at the console.

`Fits`(*args, **kwargs)	The FITS standard unit format.
`Latex`(*args, **kwargs)	Output LaTeX to display the unit based on IAU style guidelines.
`LatexInline`(*args, **kwargs)	Output LaTeX to display the unit based on IAU style guidelines with negative powers.
`OGIP`(*args, **kwargs)	Support the units in Office of Guest Investigator Programs (OGIP) FITS files.
`Unicode`(*args, **kwargs)	Output-only format to display pretty formatting at the console using Unicode characters.
`Unscaled`(*args, **kwargs)	A format that doesn't display the scale part of the unit, other than that, it is identical to the `Generic` format.
`VOUnit`(*args, **kwargs)	The IVOA standard for units used by the VO.

Class Inheritance Diagram



astropy.units.si Module

This package defines the SI units. They are also available in the `astropy.units` namespace.

Available Units

Unit	Description	Represents	Aliases	SI Prefixes
`a`	annum (a)	\(\mathrm{365.25\,d}\)	`annum`	Yes
`A`	ampere: base unit of electric current in SI		`ampere`, `amp`	Yes
`Angstrom`	ångström: 10 ** -10 m	\(\mathrm{0.1\,nm}\)	`AA`, `angstrom`	No
`arcmin`	arc minute: angular measurement	\(\mathrm{0.016666667 \,{}^{\circ}}\)	`arcminute`	Yes
`arcsec`	arc second: angular measurement	\(\mathrm{0.00027777778 \,{}^{\circ}}\)	`arcsecond`	Yes
`Bq`	becquerel: unit of radioactivity	\(\mathrm{\frac{1}{s}}\)	`becquerel`	No
`C`	coulomb: electric charge	\(\mathrm{A\,s}\)	`coulomb`	Yes

Unit	Description	Represents	Aliases	SI Prefixes
`cd`	candela: base unit of luminous intensity in SI		`candela`	Yes
`Ci`	curie: unit of radioactivity	\(\mathrm{3.7 \times 10^{10}\,Bq}\)	`curie`	No
`d`	day (d)	\(\mathrm{24\,h}\)	`day`	Yes
`deg`	degree: angular measurement 1/360 of full rotation	\(\mathrm{0.017453293\,rad}\)	`degree`	Yes
`deg_C`	Degrees Celsius		`Celsius`	No
`eV`	Electron Volt	\(\mathrm{1.6021766 \times 10^{-19}\,J}\)	`electronvolt`	Yes
`F`	Farad: electrical capacitance	\(\mathrm{\frac{C}{V}}\)	`Farad` , `farad`	Yes
`fortnight`	fortnight	\(\mathrm{2\,wk}\)		No
`g`	gram (g)	\(\mathrm{0.001\,kg}\)	`gram`	Yes
`h`	hour (h)	\(\mathrm{3600\,s}\)	`hour` , `hr`	Yes
`H`	Henry: inductance	\(\mathrm{\frac{Wb}{A}}\)	`Henry` , `henry`	Yes
`hourangle`	hour angle: angular measurement with 24 in a full circle	\(\mathrm{15\,{}^{\circ}}\)		No
`Hz`	Frequency	\(\mathrm{\frac{1}{s}}\)	`Hertz` , `hertz`	Yes
`J`	Joule: energy	\(\mathrm{N\,m}\)	`Joule` , `joule`	Yes
`K`	Kelvin: temperature with a null point at absolute zero.		`Kelvin`	Yes
`kg`	kilogram: base unit of mass in SI.		`kilogram`	No
`l`	liter: metric unit of volume	\(\mathrm{1000\,cm^{3}}\)	`L` , `liter`	Yes
`lm`	lumen: luminous flux	\(\mathrm{cd\,sr}\)	`lumen`	Yes
`lx`	lux: luminous emittance	\(\mathrm{\frac{lm}{m^{2}}}\)	`lux`	Yes
`m`	meter: base unit of length in SI		`meter`	Yes
`mas`	milli arc second: angular measurement	\(\mathrm{0.001\,{}^{\prime \prime}}\)		No
`micron`	micron: alias for micrometer (um)	\(\mathrm{\mu m}\)		No

Unit	Description	Represents	Aliases	SI Prefixes
`min`	minute (min)	\(\mathrm{60\,s}\)	`minute`	Yes
`mol`	mole: amount of a chemical substance in SI.		`mole`	Yes
`N`	Newton: force	\(\mathrm{\frac{kg\,m}{s^{2}}}\)	`Newton` , `newton`	Yes
`Ohm`	Ohm: electrical resistance	\(\mathrm{\frac{V}{A}}\)	`ohm`	Yes
`Pa`	Pascal: pressure	\(\mathrm{\frac{J}{m^{3}}}\)	`Pascal` , `pascal`	Yes
`%`	percent: one hundredth of unity, factor 0.01	\(\mathrm{0.01\,}\)	`pct`	No
`rad`	radian: angular measurement of the ratio between the length on an arc and its radius		`radian`	Yes
`s`	second: base unit of time in SI.		`second`	Yes
`S`	Siemens: electrical conductance	\(\mathrm{\frac{A}{V}}\)	`Siemens` , `siemens`	Yes
`sday`	Sidereal day (sday) is the time of one rotation of the Earth.	\(\mathrm{86164.091\,s}\)		No
`sr`	steradian: unit of solid angle in SI	\(\mathrm{rad^{2}}\)	`steradian`	Yes
`t`	Metric tonne	\(\mathrm{1000\,kg}\)	`tonne`	No
`T`	Tesla: magnetic flux density	\(\mathrm{\frac{Wb}{m^{2}}}\)	`Tesla` , `tesla`	Yes
`uas`	micro arc second: angular measurement	\(\mathrm{1 \times 10^{-6}\,{}^{\prime\prime}}\)		No
`V`	Volt: electric potential or electromotive force	\(\mathrm{\frac{J}{C}}\)	`Volt` , `volt`	Yes
`W`	Watt: power	\(\mathrm{\frac{J}{s}}\)	`Watt` , `watt`	Yes
`Wb`	Weber: magnetic flux	\(\mathrm{V\,s}\)	`Weber` , `weber`	Yes
`wk`	week (wk)	\(\mathrm{7\,d}\)	`week`	No
`yr`	year (yr)	\(\mathrm{365.25\,d}\)	`year`	Yes

astropy.units.cgs Module

This package defines the CGS units. They are also available in the top-level `astropy.units` namespace.

Available Units

Unit	Description	Represents	Aliases	SI Prefixes
`abC`	abcoulomb: CGS (EMU) of charge	\(\mathrm{Bi\,s}\)	`abcoulomb`	No
`Ba`	Barye: CGS unit of pressure	\(\mathrm{\frac{g}{cm\,s^{2}}}\)	`Barye`, `barye`	Yes
`Bi`	Biot: CGS (EMU) unit of current	\(\mathrm{\frac{cm^{1/2}\,g^{1/2}}{s}}\)	`Biot`, `abA`, `abampere`	No
`C`	coulomb: electric charge	\(\mathrm{A\,s}\)	`coulomb`	No
`cd`	candela: base unit of luminous intensity in SI		`candela`	No
`cm`	centimeter (cm)	\(\mathrm{cm}\)	`centimeter`	No
`D`	Debye: CGS unit of electric dipole moment	\(\mathrm{3.3333333 \times 10^{-30}\,C\,m}\)	`Debye`, `debye`	Yes
`deg_C`	Degrees Celsius		`Celsius`	No
`dyn`	dyne: CGS unit of force	\(\mathrm{\frac{cm\,g}{s^{2}}}\)	`dyne`	Yes
`erg`	erg: CGS unit of energy	\(\mathrm{\frac{cm^{2} \,g}{s^{2}}}\)		Yes
`Fr`	Franklin: CGS (ESU) unit of charge	\(\mathrm{\frac{cm^{3/2}\,g^{1/2}}{s}}\)	`Franklin`, `statcoulomb`, `statC`, `esu`	No
`g`	gram (g)	\(\mathrm{0.001\,kg}\)	`gram`	No
`G`	Gauss: CGS unit for magnetic field	\(\mathrm{0.0001\,T}\)	`Gauss`, `gauss`	Yes
`Gal`	Gal: CGS unit of acceleration	\(\mathrm{\frac{cm}{s^{2}}}\)	`gal`	Yes
`K`	Kelvin: temperature with a null point at absolute zero.		`Kelvin`	No
`k`	kayser: CGS unit of wavenumber	\(\mathrm{\frac{1}{cm}}\)	`Kayser`, `kayser`	Yes
`mol`	mole: amount of a chemical substance in SI.		`mole`	No
`P`	poise: CGS unit of dynamic viscosity	\(\mathrm{\frac{g}{cm\,s}}\)	`poise`	Yes
`rad`	radian: angular measurement of the ratio between the length on an arc and its radius		`radian`	No

Unit	Description	Represents	Aliases	SI Prefixes
`s`	second: base unit of time in SI.		`second`	No
`sr`	steradian: unit of solid angle in SI	\(\mathrm{rad^{2}}\)	`steradian`	No
`St`	stokes: CGS unit of kinematic viscosity	\(\mathrm{\frac{cm^{2}}{s}}\)	`stokes`	Yes
`statA`	statampere: CGS (ESU) unit of current	\(\mathrm{\frac{Fr}{s}}\)	`statampere`	No

astropy.units.astrophys Module

This package defines the astrophysics-specific units. They are also available in the `astropy.units` namespace.

Available Units

Unit	Description	Represents	Aliases	SI Prefixes
`adu`	adu			Yes
`AU`	astronomical unit: approximately the mean Earth–Sun distance.	\(\mathrm{1.4959787 \times 10^{11}\,m}\)	`au` , `astronomical_unit`	Yes
`bar`	bar: pressure	\(\mathrm{100000\,Pa}\)		Yes
`barn`	barn: unit of area used in HEP	\(\mathrm{1 \times 10^{-28}\,m^{2}}\)	`barn`	Yes
`beam`	beam			Yes
`bin`	bin			Yes
`bit`	b (bit)		`b`	Yes
`byte`	B (byte)	\(\mathrm{8\,bit}\)	`B`	Yes
`chan`	chan			Yes
`ct`	count (ct)		`count`	Yes
`cycle`	cycle: angular measurement, a full turn or rotation	\(\mathrm{6.2831853\,rad}\)	`cy`	No
`earthMass`	Earth mass	\(\mathrm{5.9721679 \times 10^{24}\,kg}\)	`M_earth` , `Mearth`	No
`earthRad`	Earth radius	\(\mathrm{6378100\,m}\)	`R_earth` , `Rearth`	No
`electron`	Number of electrons			No

Unit	Description	Represents	Aliases	SI Prefixes
`jupiterMass`	Jupiter mass	\(\mathrm{1.8981246 \times 10^{27}\,kg}\)	`M_jup` , `Mjup` , `M_jupiter` , `Mjupiter`	No
`jupiterRad`	Jupiter radius	\(\mathrm{71492000\,m}\)	`R_jup` , `Rjup` , `R_jupiter` , `Rjupiter`	No
`Jy`	Jansky: spectral flux density	\(\mathrm{1 \times 10^{-26}\,\frac{W}{Hz\,m^{2}}}\)	`Jansky` , `jansky`	Yes
`littleh`	Reduced/”dimensionless” Hubble constant			No
`lyr`	Light year	\(\mathrm{9.4607305 \times 10^{15}\,m}\)	`lightyear`	Yes
`M_e`	Electron mass	\(\mathrm{9.1093837 \times 10^{-31}\,kg}\)		No
`M_p`	Proton mass	\(\mathrm{1.6726219 \times 10^{-27}\,kg}\)		No
`pc`	parsec: approximately 3.26 light-years.	\(\mathrm{3.0856776 \times 10^{16}\,m}\)	`parsec`	Yes
`ph`	photon (ph)		`photon`	Yes
`pix`	pixel (pix)		`pixel`	Yes
`R`	Rayleigh: photon flux	\(\mathrm{7.9577472 \times 10^{8}\,\frac{ph}{s\,sr \,m^{2}}}\)	`Rayleigh` , `rayleigh`	Yes
`Ry`	Rydberg: Energy of a photon whose wavenumber is the Rydberg constant	\(\mathrm{13.605693\,eV}\)	`rydberg`	Yes
`solLum`	Solar luminance	\(\mathrm{3.828 \times 10^{26}\,W}\)	`L_sun` , `Lsun`	No
`solMass`	Solar mass	\(\mathrm{1.9884099 \times 10^{30}\,kg}\)	`M_sun` , `Msun`	No
`solRad`	Solar radius	\(\mathrm{6.957 \times 10^{8}\,m}\)	`R_sun` , `Rsun`	No
`spat`	spat: the solid angle of the sphere, 4pi sr	\(\mathrm{12.566371\,sr}\)	`sp`	No
`Sun`	Sun			No
`Torr`	Unit of pressure based on an absolute scale, now defined as	\(\mathrm{133.32237\,Pa}\)	`torr`	Yes

Unit	Description	Represents	Aliases	SI Prefixes
	exactly 1/760 of a standard atmosphere			
`u`	Unified atomic mass unit	\(\mathrm{1.6605391 \times 10^{-27}\,kg}\)	`Da` , `Dalton`	Yes
`vox`	voxel (vox)		`voxel`	Yes

astropy.units.function.units Module

This package defines units that can also be used as functions of other units. If called, their arguments are used to initialize the corresponding function unit (e.g., `u.mag(u.ct/u.s)`). Note that the prefixed versions cannot be called, as it would be unclear what, e.g., `u.mmag(u.ct/u.s)` would mean.

Available Units

Unit	Description	Represents	Aliases	SI Prefixes
`dB`	Decibel: ten per base 10 logarithmic unit	\(\mathrm{0.1\,dex}\)	`decibel`	No
`dex`	Dex: Base 10 logarithmic unit			No
`mag`	Astronomical magnitude: -2.5 per base 10 logarithmic unit	\(\mathrm{-0.4\,dex}\)		Yes

astropy.units.photometric Module

This module defines magnitude zero points and related photometric quantities.

The corresponding magnitudes are given in the description of each unit (the actual definitions are in `logarithmic`).

Available Units

Unit	Description	Represents	Aliases	SI Prefixes
`AB`	AB magnitude zero flux density (magnitude `ABmag`).	\(\mathrm{3.6307805 \times 10^{-20}\,\frac{erg}{Hz\,s \,cm^{2}}}\)	`ABflux`	No
`Bol`	Luminosity corresponding to absolute bolometric magnitude zero (magnitude `M_bol`).	\(\mathrm{3.0128 \times 10^{28}\,W}\)	`L_bol`	No
`bol`	Irradiance corresponding to appparent bolometric magnitude zero (magnitude	\(\mathrm{2.3975101 \times 10^{25}\,\frac{W}{pc^{2}}}\)	`f_bol`	No

Unit	Description	Represents	Aliases	SI Prefixes
	m_bol).			
mgy	Maggies - a linear flux unit that is the flux for a mag=0 object.To tie this onto a specific calibrated unit system, the zero_point_flux equivalency should be used.		maggy	Yes
ST	ST magnitude zero flux density (magnitude STmag).	\(\mathrm{3.6307805 \times 10^{-9}\,\frac{erg}{\mathring{A}\,s\,cm^{2}}}\)	STflux	No

Functions

zero_point_flux(flux0)	An equivalency for converting linear flux units ("maggys") defined relative to a standard source into a standardized system.

astropy.units.imperial Module

This package defines colloquially used Imperial units. They are available in the astropy.units.imperial namespace, but not in the top-level astropy.units namespace, e.g.:

```
>>> import astropy.units as u
>>> mph = u.imperial.mile / u.hour
>>> mph
Unit("mi / h")
```

To include them in compose and the results of find_equivalent_units, do:

```
>>> import astropy.units as u
>>> u.imperial.enable()
```

Available Units

Unit	Description	Represents	Aliases	SI Prefixes
ac	International acre	\(\mathrm{43560\,ft^{2}}\)	acre	No
BTU	British thermal unit	\(\mathrm{1.0550559\,kJ}\)	btu	No

Unit	Description	Represents	Aliases	SI Prefixes
`cal`	Thermochemical calorie: pre-SI metric unit of energy	\(\mathrm{4.184\,J}\)	`calorie`	No
`cup`	U.S.	\(\mathrm{0.5\,pint}\)		No
`deg_F`	Degrees Fahrenheit		`Fahrenheit`	No
`deg_R`	Rankine scale: absolute scale of thermodynamic temperature		`Rankine`	No
`foz`	U.S.	\(\mathrm{0.125\,cup}\)	`fluid_oz` , `fluid_ounce`	No
`ft`	International foot	\(\mathrm{12\,inch}\)	`foot`	No
`fur`	Furlong	\(\mathrm{660\,ft}\)	`furlong`	No
`gallon`	U.S.	\(\mathrm{3.7854118 \,\mathcal{l}}\)		No
`hp`	Electrical horsepower	\(\mathrm{745.69987\,W}\)	`horsepower`	No
`inch`	International inch	\(\mathrm{2.54\,cm}\)		No
`kcal`	Calorie: colloquial definition of Calorie	\(\mathrm{1000\,cal}\)	`Cal` , `Calorie` , `kilocal` , `kilocalorie`	No
`kip`	Kilopound: force	\(\mathrm{1000\,lbf}\)	`kilopound`	No
`kn`	nautical unit of speed: 1 nmi per hour	\(\mathrm{\frac{nmi}{h}}\)	`kt` , `knot` , `NMPH`	No
`lb`	International avoirdupois pound: mass	\(\mathrm{16\,oz}\)	`lbm` , `pound`	No
`lbf`	Pound: force	\(\mathrm{\frac{ft\,slug}{s^{2}}}\)		No
`mi`	International mile	\(\mathrm{5280\,ft}\)	`mile`	No
`mil`	Thousandth of an inch	\(\mathrm{0.001\,inch}\)	`thou`	No
`nmi`	Nautical mile	\(\mathrm{1852\,m}\)	`nauticalmile` , `NM`	No
`oz`	International avoirdupois ounce: mass	\(\mathrm{28.349523\,g}\)	`ounce`	No
`pint`	U.S.	\(\mathrm{0.5\,quart}\)		No
`psi`	Pound per square inch: pressure	\(\mathrm{\frac{lbf}{inch^{2}}}\)		No
`quart`	U.S.	\(\mathrm{0.25\,gallon}\)		No
`slug`	slug: mass	\(\mathrm{32.174049\,lb}\)		No

Unit	Description	Represents	Aliases	SI Prefixes
`st`	International avoirdupois stone: mass	\(\mathrm{14\,lb}\)	`stone`	No
`tbsp`	U.S.	\(\mathrm{0.5\,foz}\)	`tablespoon`	No
`ton`	International avoirdupois ton: mass	\(\mathrm{2000\,lb}\)		No
`tsp`	U.S.	\(\mathrm{0.33333333\,tbsp}\)	`teaspoon`	No
`yd`	International yard	\(\mathrm{3\,ft}\)	`yard`	No

Functions

`enable()`	Enable Imperial units so they appear in results of `find_equivalent_units` and `compose`.

astropy.units.cds Module

This package defines units used in the CDS format, both the units defined in Centre de Données astronomiques de Strasbourg Standards for Astronomical Catalogues 2.0 format and the complete set of supported units. This format is used by VOTable up to version 1.2.

These units are not available in the top-level `astropy.units` namespace. To use these units, you must import the `astropy.units.cds` module:

```
>>> from astropy.units import cds
>>> q = 10. * cds.lyr
```

To include them in `compose` and the results of `find_equivalent_units`, do:

```
>>> from astropy.units import cds
>>> cds.enable()
```

Available Units

Unit	Description	Represents	Aliases	SI Prefixes
`%`	percent	\(\mathrm{\%}\)		No
`---`	dimensionless and unscaled	\(\mathrm{}\)		No
`\h`	Planck constant	\(\mathrm{6.6260701 \times 10^{-34}\,J\,s}\)		Yes

Unit	Description	Represents	Aliases	SI Prefixes
`A`	Ampere	\(\mathrm{A}\)		Yes
`a`	year	\(\mathrm{a}\)		Yes
`a0`	Bohr radius	\(\mathrm{5.2917721 \times 10^{-11}\,m}\)		Yes
`AA`	Angstrom	\(\mathrm{\mathring{A}}\)	`Å`, `Angstrom`, `Angstroem`	Yes
`al`	Light year	\(\mathrm{lyr}\)		Yes
`alpha`	Fine structure constant	\(\mathrm{0.0072973526\,}\)		Yes
`arcmin`	minute of arc	\(\mathrm{{}^{\prime}}\)	`arcm`	Yes
`arcsec`	second of arc	\(\mathrm{{}^{\prime\prime}}\)	`arcs`	Yes
`atm`	atmosphere	\(\mathrm{101325\,Pa}\)		Yes
`AU`	astronomical unit	\(\mathrm{AU}\)	`au`	Yes
`bar`	bar	\(\mathrm{bar}\)		Yes
`barn`	barn	\(\mathrm{barn}\)		Yes
`bit`	bit	\(\mathrm{bit}\)		Yes
`byte`	byte	\(\mathrm{byte}\)		Yes
`C`	Coulomb	\(\mathrm{C}\)		Yes
`c`	speed of light	\(\mathrm{2.9979246 \times 10^{8}\,\frac{m}{s}}\)		Yes
`cal`	calorie	\(\mathrm{4.1854\,J}\)		Yes
`cd`	candela	\(\mathrm{cd}\)		Yes
`Crab`	Crab (X-ray) flux			Yes
`ct`	count	\(\mathrm{ct}\)		Yes
`D`	Debye (dipole)	\(\mathrm{D}\)		Yes
`d`	Julian day	\(\mathrm{d}\)		Yes
`deg`	degree	\(\mathrm{{}^{\circ}}\)	`°`, `degree`	Yes
`dyn`	dyne	\(\mathrm{dyn}\)		Yes
`e`	electron charge	\(\mathrm{1.6021766 \times 10^{-19}\,C}\)		Yes
`eps0`	electric constant	\(\mathrm{8.8541878 \times 10^{-12}\,\frac{F}{m}}\)		Yes

Unit	Description	Represents	Aliases	SI Prefixes
`erg`	erg	\(\mathrm{erg}\)		Yes
`eV`	electron volt	\(\mathrm{eV}\)		Yes
`F`	Farad	\(\mathrm{F}\)		Yes
`G`	Gravitation constant	\(\mathrm{6.6743 \times 10^{-11}\,\frac{m^{3}}{kg\,s^{2}}}\)		Yes
`g`	gram	\(\mathrm{g}\)		Yes
`gauss`	Gauss	\(\mathrm{G}\)		Yes
`geoMass`	Earth mass	\(\mathrm{M_{\oplus}}\)	`Mgeo`	Yes
`H`	Henry	\(\mathrm{H}\)		Yes
`h`	hour	\(\mathrm{h}\)		Yes
`hr`	hour	\(\mathrm{h}\)		Yes
`Hz`	Hertz	\(\mathrm{Hz}\)		Yes
`inch`	inch	\(\mathrm{0.0254\,m}\)		Yes
`J`	Joule	\(\mathrm{J}\)		Yes
`JD`	Julian day	\(\mathrm{d}\)		Yes
`jovMass`	Jupiter mass	\(\mathrm{M_{\rm J}}\)	`Mjup`	Yes
`Jy`	Jansky	\(\mathrm{Jy}\)		Yes
`K`	Kelvin	\(\mathrm{K}\)		Yes
`k`	Boltzmann	\(\mathrm{1.380649 \times 10^{-23}\,\frac{J}{K}}\)		Yes
`l`	litre	\(\mathrm{\mathcal{l}}\)		Yes
`lm`	lumen	\(\mathrm{lm}\)		Yes
`Lsun`	solar luminosity	\(\mathrm{L_{\odot}}\)	`solLum`	Yes
`lx`	lux	\(\mathrm{lx}\)		Yes
`lyr`	Light year	\(\mathrm{lyr}\)		Yes
`m`	meter	\(\mathrm{m}\)		Yes
`mag`	magnitude	\(\mathrm{mag}\)		Yes
`mas`	millisecond of arc	\(\mathrm{marcsec}\)		No
`me`	electron mass	\(\mathrm{9.1093837 \times 10^{-31}\,kg}\)		Yes

Unit	Description	Represents	Aliases	SI Prefixes
`min`	minute	\(\mathrm{min}\)		Yes
`MJD`	Julian day	\(\mathrm{d}\)		Yes
`mmHg`	millimeter of mercury	\(\mathrm{133.32239\,Pa}\)		Yes
`mol`	mole	\(\mathrm{mol}\)		Yes
`mp`	proton mass	\(\mathrm{1.6726219 \times 10^{-27}\,kg}\)		Yes
`Msun`	solar mass	\(\mathrm{M_{\odot}}\)	`solMass`	Yes
`mu0`	magnetic constant	\(\mathrm{1.2566371 \times 10^{-6}\,\frac{N}{A^{2}}}\)	`µ0`	Yes
`muB`	Bohr magneton	\(\mathrm{9.2740101 \times 10^{-24}\,\frac{J}{T}}\)		Yes
`N`	Newton	\(\mathrm{N}\)		Yes
`Ohm`	Ohm	\(\mathrm{\Omega}\)		Yes
`Pa`	Pascal	\(\mathrm{Pa}\)		Yes
`pc`	parsec	\(\mathrm{pc}\)		Yes
`ph`	photon	\(\mathrm{ph}\)		Yes
`pi`	π	\(\mathrm{3.1415927\,}\)		Yes
`pix`	pixel	\(\mathrm{pix}\)		Yes
`ppm`	parts per million	\(\mathrm{1 \times 10^{-6}\,}\)		Yes
`R`	gas constant	\(\mathrm{8.3144626\,\frac{J}{K\,mol}}\)		Yes
`rad`	radian	\(\mathrm{rad}\)		Yes
`Rgeo`	Earth equatorial radius	\(\mathrm{6378100\,m}\)		Yes
`Rjup`	Jupiter equatorial radius	\(\mathrm{71492000\,m}\)		Yes
`Rsun`	solar radius	\(\mathrm{R_{\odot}}\)	`solRad`	Yes
`Ry`	Rydberg	\(\mathrm{R_{\infty}}\)		Yes
`S`	Siemens	\(\mathrm{S}\)		Yes
`s`	second	\(\mathrm{s}\)	`sec`	Yes
`sr`	steradian	\(\mathrm{sr}\)		Yes
`Sun`	solar unit	\(\mathrm{Sun}\)		Yes
`T`	Tesla	\(\mathrm{T}\)		Yes

Unit	Description	Represents	Aliases	SI Prefixes
`t`	metric tonne	\(\mathrm{1000\,kg}\)		Yes
`u`	atomic mass	\(\mathrm{1.6605391 \times 10^{-27}\,kg}\)		Yes
`V`	Volt	\(\mathrm{V}\)		Yes
`W`	Watt	\(\mathrm{W}\)		Yes
`Wb`	Weber	\(\mathrm{Wb}\)		Yes
`yr`	year	\(\mathrm{a}\)		Yes
`µas`	microsecond of arc	\(\mathrm{\mu arcsec}\)		No

Functions

`enable`()	Enable CDS units so they appear in results of `find_equivalent_units` and `compose`.

astropy.units.equivalencies Module

A set of standard astronomical equivalencies.

Functions

`parallax`()	Returns a list of equivalence pairs that handle the conversion between parallax angle and distance.
`spectral`()	Returns a list of equivalence pairs that handle spectral wavelength, wave number, frequency, and energy equivalences.
`spectral_density`(wav[, factor])	Returns a list of equivalence pairs that handle spectral density with regard to wavelength and frequency.
`doppler_radio`(rest)	Return the equivalency pairs for the radio convention for velocity.
`doppler_optical`(rest)	Return the equivalency pairs for the optical convention for velocity.
`doppler_relativistic`(rest)	Return the equivalency pairs for the relativistic convention for velocity.
`mass_energy`()	Returns a list of equivalence pairs that handle the conversion between mass and energy.
`brightness_temperature`(frequency[, beam_area])	Defines the conversion between Jy/sr and “brightness temperature”, \(T_B\), in Kelvins.

thermodynamic_temperature(frequency[, T_cmb])	Defines the conversion between Jy/sr and “thermodynamic temperature”, \(T_{CMB}\), in Kelvins.
beam_angular_area(beam_area)	Convert between the `beam` unit, which is commonly used to express the area of a radio telescope resolution element, and an area on the sky.
dimensionless_angles()	Allow angles to be equivalent to dimensionless (with 1 rad = 1 m/m = 1).
logarithmic()	Allow logarithmic units to be converted to dimensionless fractions
temperature()	Convert between Kelvin, Celsius, Rankine and Fahrenheit here because Unit and CompositeUnit cannot do addition or subtraction properly.
temperature_energy()	Convert between Kelvin and keV(eV) to an equivalent amount.
molar_mass_amu()	Returns the equivalence between amu and molar mass.
pixel_scale(pixscale)	Convert between pixel distances (in units of `pix`) and other units, given a particular `pixscale`.
plate_scale(platescale)	Convert between lengths (to be interpreted as lengths in the focal plane) and angular units with a specified `platescale`.
with_H0([H0])	Convert between quantities with little-h and the equivalent physical units.

astropy.units.function Package

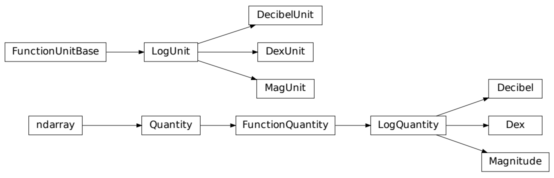
This subpackage contains classes and functions for defining and converting between different function units and quantities, i.e., using units which are some function of a physical unit, such as magnitudes and decibels.

Classes

Decibel(value[, unit, dtype, copy, order, …])	
DecibelUnit([physical_unit, function_unit])	Logarithmic physical units expressed in dB
Dex(value[, unit, dtype, copy, order, …])	
DexUnit([physical_unit, function_unit])	Logarithmic physical units expressed in magnitudes
FunctionQuantity(value[, unit, dtype, copy, …])	A representation of a (scaled) function of a number with a unit.
FunctionUnitBase([physical_unit, function_unit])	Abstract base class for function units.
LogQuantity(value[, unit, dtype, copy, …])	A representation of a (scaled) logarithm of a number with a unit
LogUnit([physical_unit, function_unit])	Logarithmic unit containing a physical one

MagUnit([physical_unit, function_unit])	Logarithmic physical units expressed in magnitudes
Magnitude(value[, unit, dtype, copy, order, …])	

Class Inheritance Diagram



astropy.units.function.logarithmic Module

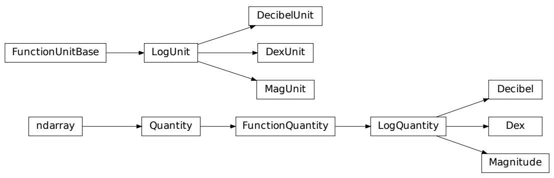
Classes

LogUnit([physical_unit, function_unit])	Logarithmic unit containing a physical one
MagUnit([physical_unit, function_unit])	Logarithmic physical units expressed in magnitudes
DexUnit([physical_unit, function_unit])	Logarithmic physical units expressed in magnitudes
DecibelUnit([physical_unit, function_unit])	Logarithmic physical units expressed in dB
LogQuantity(value[, unit, dtype, copy, …])	A representation of a (scaled) logarithm of a number with a unit
Magnitude(value[, unit, dtype, copy, order, …])	
Decibel(value[, unit, dtype, copy, order, …])	
Dex(value[, unit, dtype, copy, order, …])	

Variables

STmag	ST magnitude: STmag=-21.1 corresponds to 1 erg/s/cm2/A
ABmag	AB magnitude: ABmag=-48.6 corresponds to 1 erg/s/cm2/Hz
M_bol	Absolute bolometric magnitude: M_bol=0 corresponds to L_bol0=3.0128e+28 J / s
m_bol	Apparent bolometric magnitude: m_bol=0 corresponds to f_bol0=2.51802e-08 kg / s3

Class Inheritance Diagram



astropy.units.deprecated Module

This package defines deprecated units.

These units are not available in the top-level **astropy.units** namespace. To use these units, you must import the **astropy.units.deprecated** module:

```
>>> from astropy.units import deprecated
>>> q = 10. * deprecated.emu
```

To include them in **compose** and the results of **find_equivalent_units**, do:

```
>>> from astropy.units import deprecated
>>> deprecated.enable()
```

Available Units

Unit	Description	Represents	Aliases	SI Prefixes
`emu`	Biot: CGS (EMU) unit of current	\(\mathrm{Bi}\)		No
Prefixes for `earthMass`	Earth mass prefixes	\(\mathrm{5.9721679 \times 10^{24}\,kg}\)	`M_earth` , `Mearth`	Only
Prefixes for `earthRad`	Earth radius prefixes	\(\mathrm{6378100\,m}\)	`R_earth` , `Rearth`	Only
Prefixes for `jupiterMass`	Jupiter mass prefixes	\(\mathrm{1.8981246 \times 10^{27}\,kg}\)	`M_jup` , `Mjup` , `M_jupiter` , `Mjupiter`	Only
Prefixes for `jupiterRad`	Jupiter radius prefixes	\(\mathrm{71492000\,m}\)	`R_jup` , `Rjup` , `R_jupiter` , `Rjupiter`	Only

Functions

`enable()`	Enable deprecated units so they appear in results of `find_equivalent_units` and `compose`.

astropy.units.required_by_vounit Module

This package defines SI prefixed units that are required by the VOUnit standard but that are rarely used in practice and liable to lead to confusion (such as `msolMass` for milli-solar mass). They are in a separate module from `astropy.units.deprecated` because they need to be enabled by default for `astropy.units` to parse compliant VOUnit strings. As a result, e.g., `Unit('msolMass')` will just work, but to access the unit directly, use `astropy.units.required_by_vounit.msolMass` instead of the more typical idiom possible for the non-prefixed unit, `astropy.units.solMass`.

Available Units

Unit	Description	Represents	Aliases	SI Prefixes
Prefixes for `solLum`	Solar luminance prefixes	\(\mathrm{3.828 \times 10^{26}\,W}\)	`L_sun`, `Lsun`	Only
Prefixes for `solMass`	Solar mass prefixes	\(\mathrm{1.9884099 \times 10^{30}\,kg}\)	`M_sun`, `Msun`	Only
Prefixes for `solRad`	Solar radius prefixes	\(\mathrm{6.957 \times 10^{8}\,m}\)	`R_sun`, `Rsun`	Only

N-Dimensional Datasets (`astropy.nddata`)

Introduction

The `nddata` package provides classes to represent images and other gridded data, some essential functions for manipulating images, and the infrastructure for package developers who wish to include support for the image classes.

Getting Started

NDData

The primary purpose of `NDData` is to act as a *container* for data, metadata, and other related information like a mask.

An `NDData` object can be instantiated by passing it an n-dimensional `numpy` array:

```
>>> import numpy as np
>>> from astropy.nddata import NDData
>>> array = np.zeros((12, 12, 12))  # a 3-dimensional array with all
```

```
zeros
>>> ndd1 = NDData(array)
```

Or something that can be converted to a **numpy.ndarray**:

```
>>> ndd2 = NDData([1, 2, 3, 4])
>>> ndd2
NDData([1, 2, 3, 4])
```

And can be accessed again via the `data` attribute:

```
>>> ndd2.data
array([1, 2, 3, 4])
```

It also supports additional properties like a `unit` or `mask` for the data, a `wcs` (World Coordinate System) and `uncertainty` of the data and additional `meta` attributes:

```
>>> data = np.array([1,2,3,4])
>>> mask = data > 2
>>> unit = 'erg / s'
>>> from astropy.nddata import StdDevUncertainty
>>> uncertainty = StdDevUncertainty(np.sqrt(data)) # representing
standard deviation
>>> meta = {'object': 'fictional data.'}
>>> ndd = NDData(data, mask=mask, unit=unit, uncertainty=uncertainty,
...              meta=meta)
>>> ndd
NDData([1, 2, 3, 4])
```

The representation only displays the `data`; the other attributes need to be accessed directly, for example, `ndd.mask` to access the mask.

NDDataRef

Building upon this pure container, **NDDataRef** implements:

- A `read` and `write` method to access `astropy`'s unified file I/O interface.
- Simple arithmetics like addition, subtraction, division, and multiplication.
- Slicing.

Instances are created in the same way:

```
>>> from astropy.nddata import NDDataRef
>>> ndd = NDDataRef(ndd)
>>> ndd
NDDataRef([1, 2, 3, 4])
```

But also support arithmetic (NDData Arithmetic) like addition:

```
>>> import astropy.units as u
>>> ndd2 = ndd.add([4, -3.5, 3, 2.5] * u.erg / u.s)
>>> ndd2
NDDataRef([ 5. , -1.5,  6. ,  6.5])
```

Because these operations have a wide range of options, these are not available using arithmetic operators like `+`.

Slicing or indexing (Slicing and Indexing NDData) is possible (with warnings issued if some attribute cannot be sliced):

```
>>> ndd2[2:]  # discard the first two elements
NDDataRef([6. , 6.5])
>>> ndd2[1]   # get the second element
NDDataRef(-1.5)
```

Working with Two-Dimensional Data Like Images

Though the **nddata** package supports any kind of gridded data, this introduction will focus on the use of **nddata** for two-dimensional images. To get started, we will construct a two-dimensional image with a few sources, some Gaussian noise, and a “cosmic ray” which we will later mask out.

Examples

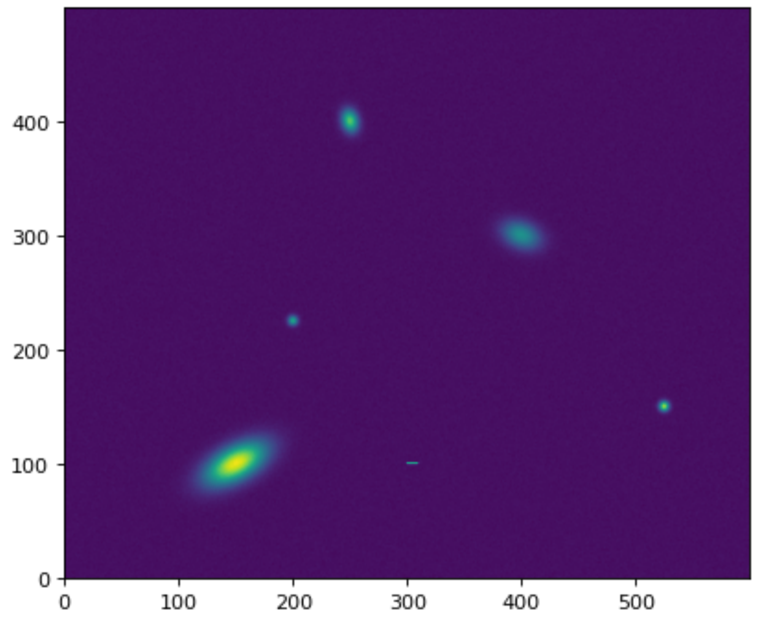
First, construct a two-dimensional image with a few sources, some Gaussian noise, and a “cosmic ray”:

```
>>> import numpy as np
>>> from astropy.modeling.models import Gaussian2D
>>> y, x = np.mgrid[0:500, 0:600]
>>> data = (Gaussian2D(1, 150, 100, 20, 10, theta=0.5)(x, y) +
...         Gaussian2D(0.5, 400, 300, 8, 12, theta=1.2)(x,y) +
...         Gaussian2D(0.75, 250, 400, 5, 7, theta=0.23)(x,y) +
...         Gaussian2D(0.9, 525, 150, 3, 3)(x,y) +
...         Gaussian2D(0.6, 200, 225, 3, 3)(x,y))
>>> data += 0.01 * np.random.randn(500, 600)
>>> cosmic_ray_value = 0.997
>>> data[100, 300:310] = cosmic_ray_value
```

This image has a large “galaxy” in the lower left and the “cosmic ray” is the horizontal line in the lower middle of the image:

```
>>> import matplotlib.pyplot as plt
>>> plt.imshow(data, origin='lower')
```

(png, svg, pdf)



The “cosmic ray” can be masked out in this test image, like this:

```
>>> mask = (data == cosmic_ray_value)
```

CCDData Class for Images

The **CCDData** object, like the other objects in this package, can store the data, a mask, and metadata. The **CCDData** object requires that a unit be specified:

```
>>> from astropy.nddata import CCDData
>>> ccd = CCDData(data, mask=mask,
...               meta={'object': 'fake galaxy', 'filter': 'R'},
...               unit='adu')
```

Slicing

Slicing works the way you would expect with the mask and, if present, WCS, sliced appropriately:

```
>>> ccd2 = ccd[:200, :]
>>> ccd2.data.shape
(200, 600)
```

```
>>> ccd2.mask.shape
(200, 600)
>>> # Show the mask in a region around the cosmic ray:
>>> ccd2.mask[99:102, 299:311]
array([[False, False, False, False, False, False, False, False,
False,
        False, False, False],
       [False,  True,  True,  True,  True,  True,  True,  True,
True,
         True,  True, False],
       [False, False, False, False, False, False, False, False,
False,
        False, False, False]]...)
```

For many applications it may be more convenient to use **`Cutout2D`**, described in image_utilities.

Image Arithmetic, Including Uncertainty

Methods are provided for basic arithmetic operations between images, including propagation of uncertainties. Three uncertainty types are supported: variance (**`VarianceUncertainty`**), standard deviation (**`StdDevUncertainty`**), and inverse variance (**`InverseVariance`**).

Examples

This example creates an uncertainty that is Poisson error, stored as a variance:

```
>>> from astropy.nddata import VarianceUncertainty
>>> poisson_noise = np.ma.sqrt(np.ma.abs(ccd.data))
>>> ccd.uncertainty = VarianceUncertainty(poisson_noise ** 2)
```

As a convenience, the uncertainty can also be set with a `numpy` array. In that case, the uncertainty is assumed to be the standard deviation:

```
>>> ccd.uncertainty = poisson_noise
INFO: array provided for uncertainty; assuming it is a
StdDevUncertainty. [astropy.nddata.ccddata]
```

If we make a copy of the image and add that to the original, the uncertainty changes as expected:

```
>>> ccd2 = ccd.copy()
>>> added_ccds = ccd.add(ccd2, handle_meta='first_found')
>>> added_ccds.uncertainty.array[0, 0] / ccd.uncertainty.array[0, 0]
/ np.sqrt(2)
0.9999999999999989
```

Reading and Writing

A **CCDData** can be saved to a FITS file:

```
>>> ccd.write('test_file.fits')
```

And can also be read in from a FITS file:

```
>>> ccd2 = CCDData.read('test_file.fits')
```

Note the unit is stored in the `BUNIT` keyword in the header on saving, and is read from the header if it is present.

Detailed help on the available keyword arguments for reading and writing can be obtained via the `help()` method as follows:

```
>>> CCDData.read.help('fits')  # Get help on the CCDData FITS reader
>>> CCDData.writer.help('fits')  # Get help on the CCDData FITS
writer
```

Image Utilities

Cutouts

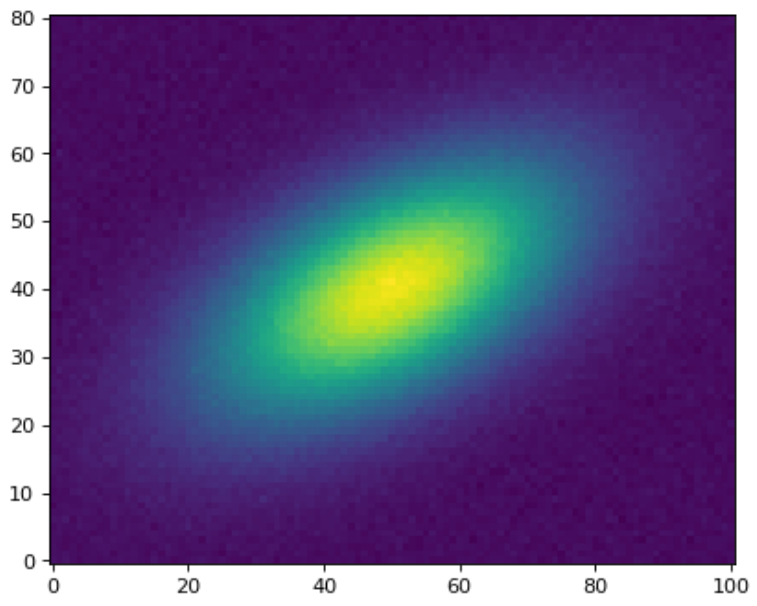
Though slicing directly is one way to extract a subframe, **Cutout2D** provides more convenient access to cutouts from the data.

Examples

This example pulls out the large "galaxy" in the lower left of the image, with the center of the cutout at `position`:

```
>>> from astropy.nddata import Cutout2D
>>> position = (149.7, 100.1)
>>> size = (81, 101)     # pixels
>>> cutout = Cutout2D(ccd, position, size)
>>> plt.imshow(cutout.data, origin='lower')
```

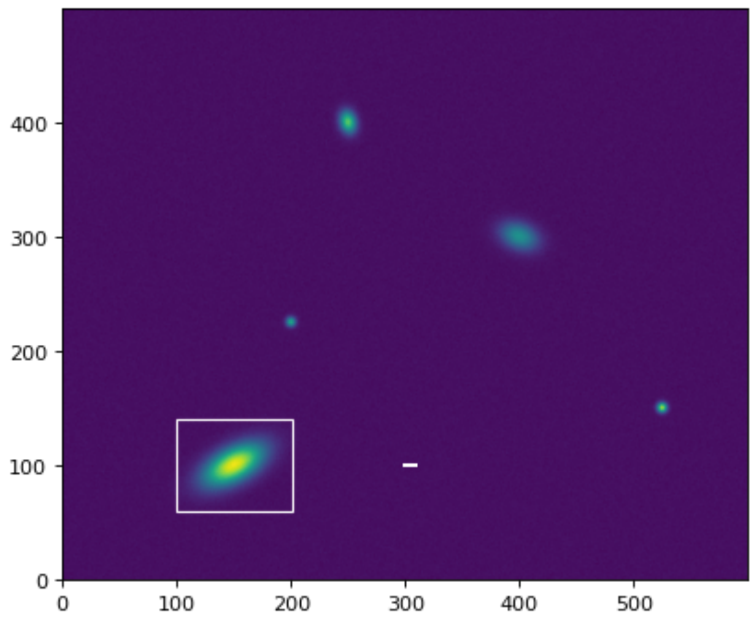
(png, svg, pdf)



This cutout can also plot itself on the original image:

```
>>> plt.imshow(ccd, origin='lower')
>>> cutout.plot_on_original(color='white')
```

(png, svg, pdf)



The cutout also provides methods for finding pixel coordinates in the original or in the cutout; recall that `position` is the center of the cutout in the original image:

```
>>> position
(149.7, 100.1)
```

```
>>> cutout.to_cutout_position(position)
(49.7, 40.099999999999994)
>>> cutout.to_original_position((49.7, 40.099999999999994))
 (149.7, 100.1)
```

For more details, including constructing a cutout from World Coordinates and the options for handling cutouts that go beyond the bounds of the original image, see 2D Cutout Images.

Image Resizing

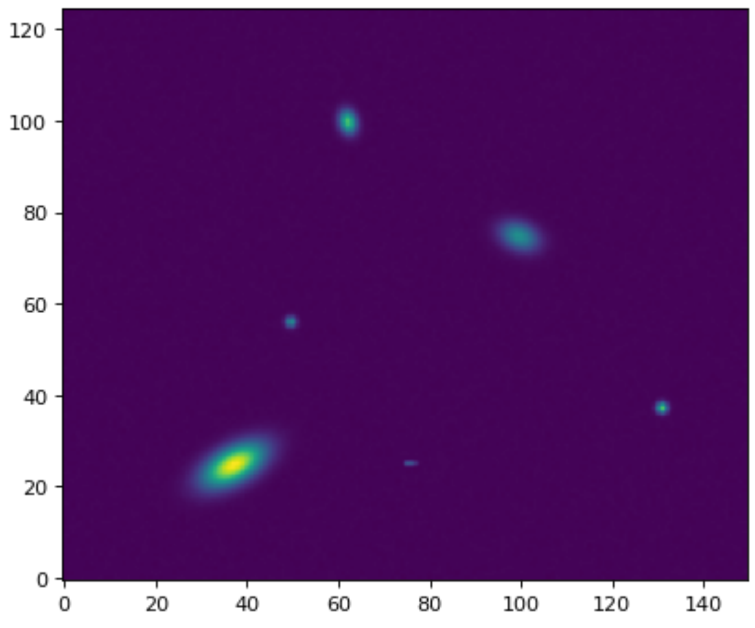
The functions **block_reduce** and **block_replicate** resize images.

Example

This example reduces the size of the image by a factor of 4. Note that the result is a **numpy.ndarray**; the mask, metadata, etc. are discarded:

```
>>> from astropy.nddata import block_reduce, block_replicate
>>> smaller = block_reduce(ccd, 4)
>>> smaller
array(...)
>>> plt.imshow(smaller, origin='lower')
```

(png, svg, pdf)



By default, both **block_reduce** and **block_replicate** conserve flux.

Other Image Classes

There are two less restrictive classes, `NDDataArray` and `NDDataRef`, that can be used to hold image data. They are primarily of interest to those who may want to create their own image class by subclassing from one of the classes in the `nddata` package. The main differences between them are:

- `NDDataRef` can be sliced and has methods for basic arithmetic operations, but the user needs to use one of the uncertainty classes to define an uncertainty. See NDDataRef for more detail. Most of its properties must be set when the object is created because they are not mutable.
- `NDDataArray` extends `NDDataRef` by adding the methods necessary for it to behave like a `numpy` array in expressions and adds setters for several properties. It lacks the ability to automatically recognize and read data from FITS files and does not attempt to automatically set the WCS property.
- `CCDData` extends `NDDataArray` by setting up a default uncertainty class, setting up straightforward read/write to FITS files, and automatically setting up a WCS property.

More General Gridded Data Classes

There are two additional classes in the `nddata` package that are of interest primarily to users who either need a custom image class that goes beyond the classes discussed so far, or who are working with gridded data that is not an image.

- `NDData` is a container class for holding general gridded data. It includes a handful of basic attributes, but no slicing or arithmetic. More information about this class is in NDData.
- `NDDataBase` is an abstract base class that developers of new gridded data classes can subclass to declare that the new class follows the `NDData` interface. More details are in Subclassing.

Additional Examples

The list of packages below that use the `nddata` framework is intended to be useful to either users writing their own image classes or those looking for an image class that goes beyond what `CCDData` does.

- The SunPy project uses `NDData` as the foundation for its Map classes.
- The class `NDDataRef` is used in specutils as the basis for Spectrum1D, which adds several methods useful for spectra.
- The package ndmapper, which makes it easy to build reduction pipelines for optical data, uses `NDDataArray` as its image object.
- The package ccdproc uses the `CCDData` class throughout for implementing optical/IR image reduction.

Using `nddata`

CCDData Class

Getting Started

Getting Data In

Creating a **`CCDData`** object from any array-like data using **`astropy.nddata`** is convenient:

```
>>> import numpy as np
>>> from astropy.nddata import CCDData
>>> ccd = CCDData(np.arange(10), unit="adu")
```

Note that behind the scenes, this creates references to (not copies of) your data when possible, so modifying the data in `ccd` will modify the underlying data.

You are **required** to provide a unit for your data. The most frequently used units for these objects are likely to be `adu`, `photon`, and `electron`, which can be set either by providing the string name of the unit (as in the example above) or from unit objects:

```
>>> from astropy import units as u
>>> ccd_photon = CCDData([1, 2, 3], unit=u.photon)
>>> ccd_electron = CCDData([1, 2, 3], unit="electron")
```

If you prefer *not* to use the unit functionality, then use the special unit `u.dimensionless_unscaled` when you create your **`CCDData`** images:

```
>>> ccd_unitless = CCDData(np.zeros((10, 10)),
...                        unit=u.dimensionless_unscaled)
```

A **`CCDData`** object can also be initialized from a FITS filename or URL:

```
>>> ccd = CCDData.read('my_file.fits', unit="adu")
>>> ccd = CCDData.read('http://data.astropy.org/tutorials/FITS-images
/HorseHead.fits', unit="adu", cache=True)
```

If there is a unit in the FITS file (in the `BUNIT` keyword), that will be used, but explicitly providing a unit in `read` will override any unit in the FITS file.

There is no restriction at all on what the unit can be — any unit in astropy.units or another that you create yourself will work.

In addition, the user can specify the extension in a FITS file to use:

```
>>> ccd = CCDData.read('my_file.fits', hdu=1, unit="adu")
```

If `hdu` is not specified, it will assume the data is in the primary extension. If there is no data in the primary extension, the first extension with image data will be used.

Metadata

When initializing from a FITS file, the `header` property is initialized using the header of the FITS file. Metadata is optional, and can be provided by any dictionary or dict-like object:

```
>>> ccd_simple = CCDData(np.arange(10), unit="adu")
>>> my_meta = {'observer': 'Edwin Hubble', 'exposure': 30.0}
>>> ccd_simple.header = my_meta  # or use ccd_simple.meta = my_meta
```

Whether the metadata is case-sensitive or not depends on how it is initialized. A FITS header, for example, is not case-sensitive, but a Python dictionary is.

Getting Data Out

A CCDData object behaves like a `numpy` array (masked if the CCDData mask is set) in expressions, and the underlying data (ignoring any mask) is accessed through the `data` attribute:

```
>>> ccd_masked = CCDData([1, 2, 3], unit="adu", mask=[0, 0, 1])
>>> 2 * np.ones(3) * ccd_masked   # one return value will be masked
masked_array(data=[2.0, 4.0, --],
             mask=[False, False,  True],
       fill_value=1e+20)
>>> 2 * np.ones(3) * ccd_masked.data   # ignores the mask
array([2., 4., 6.])
```

You can force conversion to a `numpy` array with:

```
>>> np.asarray(ccd_masked)
array([1, 2, 3])
>>> np.ma.array(ccd_masked.data, mask=ccd_masked.mask)
masked_array(data=[1, 2, --],
             mask=[False, False,  True],
       fill_value=999999)
```

A method for converting a CCDData object to a FITS HDU list is also available.

It converts the metadata to a FITS header:

```
>>> hdulist = ccd_masked.to_hdu()
```

You can also write directly to a FITS file:

```
>>> ccd_masked.write('my_image.fits')
```

Masks and Flags

Although it is not required when a **CCDData** image is created, you can also specify a mask and/or flags.

A mask is a boolean array the same size as the data in which a value of `True` indicates that a particular pixel should be masked (*i.e.*, not be included in arithmetic operations or aggregation).

Flags are one or more additional arrays (of any type) whose shape matches the shape of the data. One particularly useful type of flag is a bit planes; for more details about bit planes and the functions `astropy` provides for converting them to binary masks, see Utility Functions for Handling Bit Masks and Mask Arrays. For more details on setting flags, see **NDData**.

WCS

The `wcs` attribute of a **CCDData** object can be set two ways.

- If the **CCDData** object is created from a FITS file that has WCS keywords in the header, the `wcs` attribute is set to a **WCS** object using the information in the FITS header.
- The WCS can also be provided when the **CCDData** object is constructed with the `wcs` argument.

Either way, the `wcs` attribute is kept up to date if the **CCDData** image is trimmed.

Uncertainty

You can set the uncertainty directly, either by creating a **StdDevUncertainty** object first:

```
>>> data = np.random.normal(size=(10, 10), loc=1.0, scale=0.1)
>>> ccd = CCDData(data, unit="electron")
>>> from astropy.nddata.nduncertainty import StdDevUncertainty
>>> uncertainty = 0.1 * ccd.data  # can be any array whose shape
```

```
matches the data
>>> my_uncertainty = StdDevUncertainty(uncertainty)
>>> ccd.uncertainty = my_uncertainty
```

Or by providing a **ndarray** with the same shape as the data:

```
>>> ccd.uncertainty = 0.1 * ccd.data
INFO: array provided for uncertainty; assuming it is a
StdDevUncertainty. [...]
```

In this case, the uncertainty is assumed to be **StdDevUncertainty**.

Two other uncertainty classes are available for which error propagation is also supported: **VarianceUncertainty** and **InverseVariance**. Using one of these three uncertainties is required to enable error propagation in **CCDData**.

If you want access to the underlying uncertainty, use its `.array` attribute:

```
>>> ccd.uncertainty.array
array(...)
```

Arithmetic with Images

Methods are provided to perform arithmetic operations with a **CCDData** image and a number, an `astropy` **Quantity** (a number with units), or another **CCDData** image.

Using these methods propagates errors correctly (if the errors are uncorrelated), takes care of any necessary unit conversions, and applies masks appropriately. Note that the metadata of the result is *not* set if the operation is between two **CCDData** objects.

```
>>> result = ccd.multiply(0.2 * u.adu)
>>> uncertainty_ratio = result.uncertainty.array[0,
0]/ccd.uncertainty.array[0, 0]
>>> round(uncertainty_ratio, 5)
0.2
>>> result.unit
Unit("adu electron")
```

Note

The affiliated package ccdproc provides functions for many common data reduction operations. Those functions try to construct a sensible header for the result and provide a mechanism for logging the action of the function in the header.

The arithmetic operators `*`, `/`, `+`, and `-` are *not* overridden.

Note

If two images have different WCS values, the `wcs` on the first **CCDData** object will be used for the resultant object.

Image Utilities

Overview

The **astropy.nddata.utils** module includes general utility functions for array operations.

2D Cutout Images

Getting Started

The **Cutout2D** class can be used to create a postage stamp cutout image from a 2D array. If an optional **WCS** object is input to **Cutout2D**, then the **Cutout2D** object will contain an updated **WCS** corresponding to the cutout array.

First, we simulate a single source on a 2D data array. If you would like to simulate many sources, see Efficient Model Rendering with Bounding Boxes.

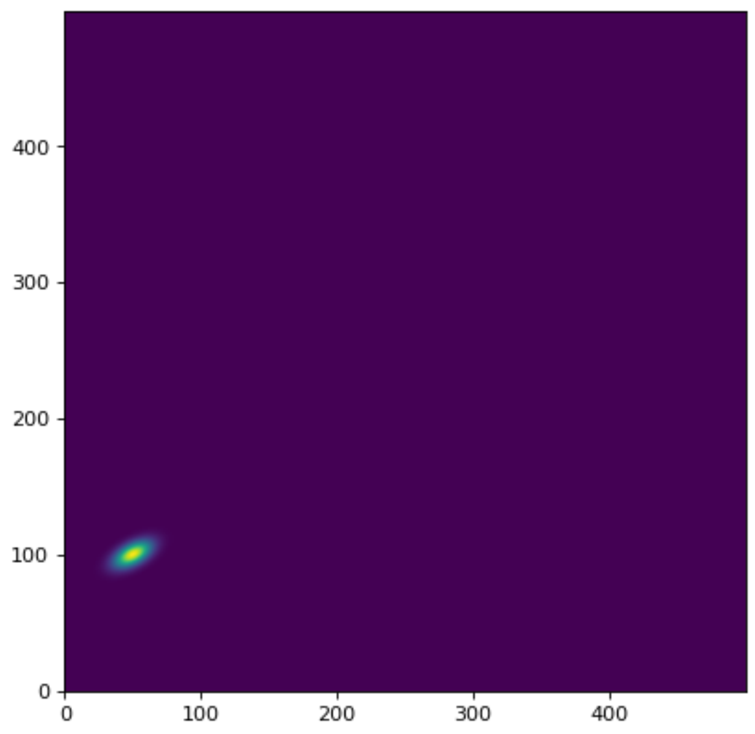
Note: The pair convention is different for **size** and **position**! The position is specified as (x,y), but the size is specified as (y,x).

```
>>> import numpy as np
>>> from astropy.modeling.models import Gaussian2D
>>> y, x = np.mgrid[0:500, 0:500]
>>> data = Gaussian2D(1, 50, 100, 10, 5, theta=0.5)(x, y)
```

Now, we can display the image:

```
>>> import matplotlib.pyplot as plt
>>> plt.imshow(data, origin='lower')
```

(png, svg, pdf)



Next we can create a cutout for the single object in this image. We create a cutout centered at position `(x, y) = (49.7, 100.1)` with a size of `(ny, nx) = (41, 51)` pixels:

```
>>> from astropy.nddata import Cutout2D
>>> from astropy import units as u
>>> position = (49.7, 100.1)
>>> size = (41, 51)      # pixels
>>> cutout = Cutout2D(data, position, size)
```

The `size` keyword can also be a **`Quantity`** object:

```
>>> size = u.Quantity((41, 51), u.pixel)
>>> cutout = Cutout2D(data, position, size)
```

Or contain **`Quantity`** objects:

```
>>> size = (41*u.pixel, 51*u.pixel)
>>> cutout = Cutout2D(data, position, size)
```

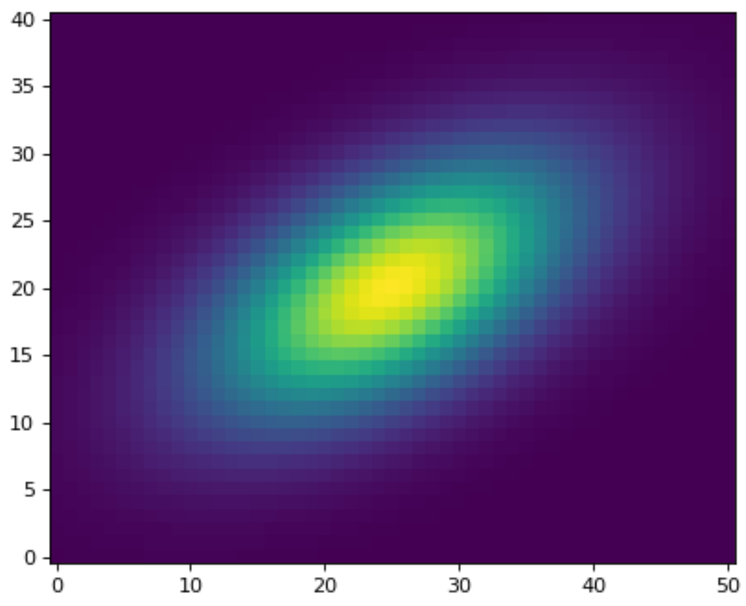
A square cutout image can be generated by passing an integer or a scalar **`Quantity`**:

```
>>> size = 41
>>> cutout2 = Cutout2D(data, position, size)
>>> size = 41 * u.pixel
>>> cutout2 = Cutout2D(data, position, size)
```

The cutout array is stored in the `data` attribute of the **`Cutout2D`** instance. If the `copy` keyword is **`False`** (default), then `cutout.data` will be a view into the original `data` array. If `copy=True`, then `cutout.data` will hold a copy of the original `data`. Now we display the cutout image:

```
>>> cutout = Cutout2D(data, position, (41, 51))
>>> plt.imshow(cutout.data, origin='lower')
```

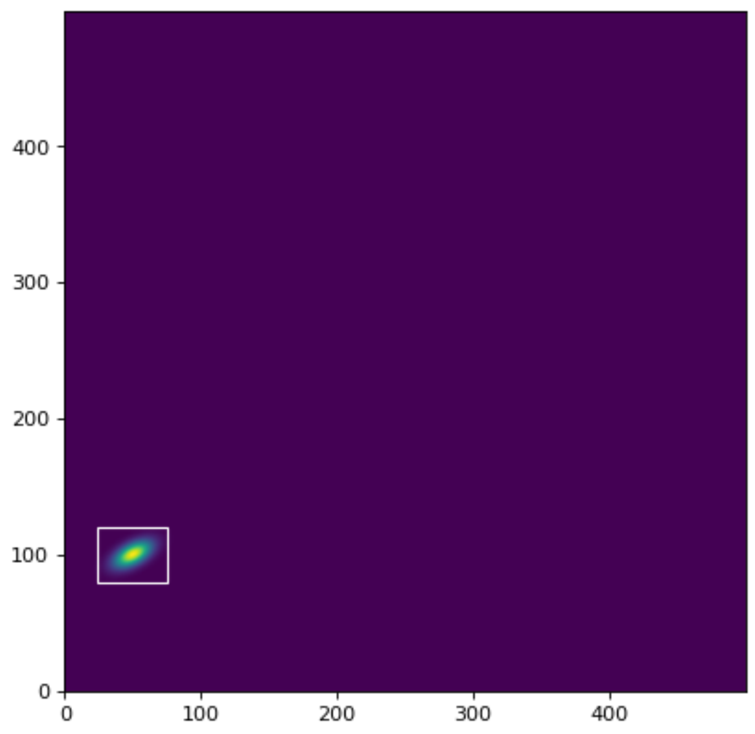
(png, svg, pdf)



The cutout object can plot its bounding box on the original data using the **`plot_on_original()`** method:

```
>>> plt.imshow(data, origin='lower')
>>> cutout.plot_on_original(color='white')
```

(png, svg, pdf)



Many properties of the cutout array are also stored as attributes, including:

```
>>> # shape of the cutout array
>>> print(cutout.shape)
(41, 51)

>>> # rounded pixel index of the input position
>>> print(cutout.position_original)
(50, 100)

>>> # corresponding position in the cutout array
>>> print(cutout.position_cutout)
(25, 20)

>>> # (non-rounded) input position in both the original and cutout
arrays
>>> print((cutout.input_position_original,
cutout.input_position_cutout))
((49.7, 100.1), (24.700000000000003, 20.099999999999994))

>>> # the origin pixel in both arrays
>>> print((cutout.origin_original, cutout.origin_cutout))
((25, 80), (0, 0))

>>> # tuple of slice objects for the original array
>>> print(cutout.slices_original)
(slice(80, 121, None), slice(25, 76, None))
```

```
>>> # tuple of slice objects for the cutout array
>>> print(cutout.slices_cutout)
(slice(0, 41, None), slice(0, 51, None))
```

There are also two **Cutout2D** methods to convert pixel positions between the original and cutout arrays:

```
>>> print(cutout.to_original_position((2, 1)))
(27, 81)

>>> print(cutout.to_cutout_position((27, 81)))
(2, 1)
```

2D Cutout Modes

There are three modes for creating cutout arrays: `'trim'`, `'partial'`, and `'strict'`. For the `'partial'` and `'trim'` modes, a partial overlap of the cutout array and the input `data` array is sufficient. For the `'strict'` mode, the cutout array has to be fully contained within the `data` array, otherwise an **PartialOverlapError** is raised. In all modes, non-overlapping arrays will raise a **NoOverlapError**. In `'partial'` mode, positions in the cutout array that do not overlap with the `data` array will be filled with `fill_value`. In `'trim'` mode only the overlapping elements are returned, thus the resulting cutout array may be smaller than the requested `size`.

The default uses `mode='trim'`, which can result in cutout arrays that are smaller than the requested `size`:

```
>>> data2 = np.arange(20.).reshape(5, 4)
>>> cutout1 = Cutout2D(data2, (0, 0), (3, 3), mode='trim')
>>> print(cutout1.data)
[[0. 1.]
 [4. 5.]]
>>> print(cutout1.shape)
(2, 2)
>>> print((cutout1.position_original, cutout1.position_cutout))
((0, 0), (0, 0))
```

With `mode='partial'`, the cutout will never be trimmed. Instead it will be filled with `fill_value` (the default is `numpy.nan`) if the cutout is not fully contained in the data array:

```
>>> cutout2 = Cutout2D(data2, (0, 0), (3, 3), mode='partial')
>>> print(cutout2.data)
[[nan nan nan]
```

```
 [nan  0.  1.]
 [nan  4.  5.]]
```

Note that for the `'partial'` mode, the positions (and several other attributes) are calculated for on the *valid* (non-filled) cutout values:

```
>>> print((cutout2.position_original, cutout2.position_cutout))
((0, 0), (1, 1))
>>> print((cutout2.origin_original, cutout2.origin_cutout))
((0, 0), (1, 1))
>>> print(cutout2.slices_original)
(slice(0, 2, None), slice(0, 2, None))
>>> print(cutout2.slices_cutout)
(slice(1, 3, None), slice(1, 3, None))
```

Using `mode='strict'` will raise an exception if the cutout is not fully contained in the data array:

```
>>> cutout3 = Cutout2D(data2, (0, 0), (3, 3), mode='strict')
PartialOverlapError: Arrays overlap only partially.
```

2D Cutout from a SkyCoord Position

The input `position` can also be specified as a **SkyCoord**, in which case a **WCS** object must be input via the `wcs` keyword.

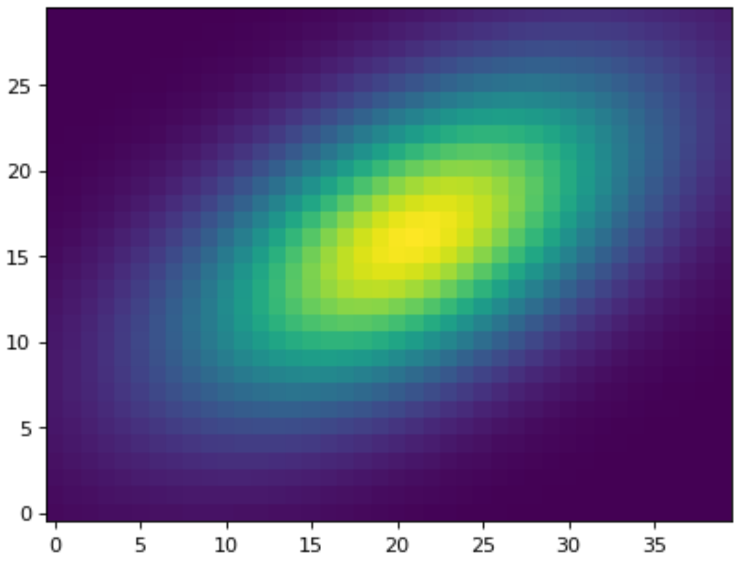
First, we define a **SkyCoord** position and a **WCS** object for our data (usually this would come from your FITS header):

```
>>> from astropy.coordinates import SkyCoord
>>> from astropy.wcs import WCS
>>> position = SkyCoord('13h11m29.96s -01d19m18.7s', frame='icrs')
>>> wcs = WCS(naxis=2)
>>> rho = np.pi / 3.
>>> scale = 0.05 / 3600.
>>> wcs.wcs.cd = [[scale*np.cos(rho), -scale*np.sin(rho)],
...               [scale*np.sin(rho), scale*np.cos(rho)]]
>>> wcs.wcs.ctype = ['RA---TAN', 'DEC--TAN']
>>> wcs.wcs.crval = [position.ra.to_value(u.deg),
...                  position.dec.to_value(u.deg)]
>>> wcs.wcs.crpix = [50, 100]
```

Now we can create the cutout array using the **SkyCoord** position and `wcs` object:

```
>>> cutout = Cutout2D(data, position, (30, 40), wcs=wcs)
>>> plt.imshow(cutout.data, origin='lower')
```

(png, svg, pdf)



The `wcs` attribute of the **Cutout2D** object now contains the propagated **WCS** for the cutout array. Now we can find the sky coordinates for a given pixel in the cutout array. Note that we need to use the `cutout.wcs` object for the cutout positions:

```
>>> from astropy.wcs.utils import pixel_to_skycoord
>>> x_cutout, y_cutout = (5, 10)
>>> pixel_to_skycoord(x_cutout, y_cutout, cutout.wcs)
<SkyCoord (ICRS): (ra, dec) in deg
    ( 197.8747893, -1.32207626)>
```

We now find the corresponding pixel in the original `data` array and its sky coordinates:

```
>>> x_data, y_data = cutout.to_original_position((x_cutout,
y_cutout))
>>> pixel_to_skycoord(x_data, y_data, wcs)
<SkyCoord (ICRS): (ra, dec) in deg
    ( 197.8747893, -1.32207626)>
```

As expected, the sky coordinates in the original `data` and the cutout array agree.

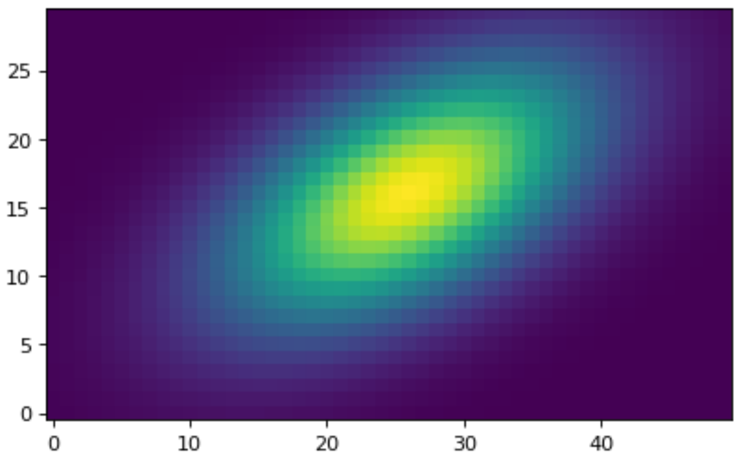
2D Cutout Using an Angular `size`

The input `size` can also be specified as a **Quantity** in angular units (e.g., degrees, arcminutes, arcseconds, etc.). For this case, a **WCS** object must be input via the `wcs` keyword.

For this example, we will use the data, **SkyCoord** position, and `wcs` object from above to create a cutout with size 1.5 x 2.5 arcseconds:

```
>>> size = u.Quantity((1.5, 2.5), u.arcsec)
>>> cutout = Cutout2D(data, position, size, wcs=wcs)
>>> plt.imshow(cutout.data, origin='lower')
```

(png, svg, pdf)



Saving a 2D Cutout to a FITS File with an Updated WCS

A **Cutout2D** object can be saved to a FITS file, including the updated WCS object for the cutout region. In this example, we download an example FITS image and create a cutout image. The resulting **Cutout2D** object is then saved to a new FITS file with the updated WCS for the cutout region.

```
# Download an example FITS file, create a 2D cutout, and save it to a
# new FITS file, including the updated cutout WCS.
from astropy.io import fits
from astropy.nddata import Cutout2D
from astropy.utils.data import download_file
from astropy.wcs import WCS


def download_image_save_cutout(url, position, size):
    # Download the image
    filename = download_file(url)

    # Load the image and the WCS
    hdu = fits.open(filename)[0]
    wcs = WCS(hdu.header)

    # Make the cutout, including the WCS
    cutout = Cutout2D(hdu.data, position=position, size=size,
wcs=wcs)

    # Put the cutout image in the FITS HDU
    hdu.data = cutout.data

    # Update the FITS header with the cutout WCS
    hdu.header.update(cutout.wcs.to_header())

    # Write the cutout to a new FITS file
    cutout_filename = 'example_cutout.fits'
    hdu.writeto(cutout_filename, overwrite=True)


if __name__ == '__main__':
    url = 'https://astropy.stsci.edu/data/photometry
/spitzer_example_image.fits'

    position = (500, 300)
    size = (400, 400)
    download_image_save_cutout(url, position, size)
```

Utility Functions for Handling Bit Masks and Mask Arrays

It is common to use bit fields, such as integer variables whose individual bits represent some attributes, to characterize the state of data. For example, Hubble Space Telescope (HST) uses arrays of bit fields to characterize data quality (DQ) of HST images. See, for example, DQ field values for WFPC2 image data (see Table 3.3) and WFC3 image data (see Table 3.3). As you can see, the meaning assigned to various *bit flags* for the two instruments is generally different.

Bit fields can be thought of as tightly packed collections of bit flags. Using masking we can “inspect” the status of individual bits.

One common operation performed on bit field arrays is their conversion to boolean masks, for example, by assigning boolean **`True`** (in the boolean mask) to those elements that correspond to non-zero-valued bit fields (bit fields with at least one bit set to `1`) or, oftentimes, by assigning **`True`** to elements whose corresponding bit fields have only *specific fields* set (to `1`). This more sophisticated analysis of bit fields can be accomplished using *bit masks* and the aforementioned masking operation.

The **`bitmask`** module provides two functions that facilitate conversion of bit field arrays (i.e., DQ arrays) to boolean masks: `bitfield_to_boolean_mask` converts an input bit field array to a boolean mask using an input bit mask (or list of individual bit flags) and `interpret_bit_flags` creates a bit mask from an input list of individual bit flags.

Creating Boolean Masks

Overview

`bitfield_to_boolean_mask` by default assumes that all input bit fields that have at least one bit turned "ON" corresponds to "bad" data (i.e., pixels) and converts them to boolean `True` in the output boolean mask (otherwise output boolean mask values are set to `False`).

Often, for specific algorithms and situations, some bit flags are okay and can be ignored. `bitfield_to_boolean_mask` accepts lists of bit flags that *by default must be ignored* in the input bit fields when creating boolean masks.

Fundamentally, *by default*, `bitfield_to_boolean_mask` performs the following operation:

```
(1)    boolean_mask = (bitfield & ~bit_mask) != 0
```

(Here `&` is bitwise `and` while `~` is the bitwise `not` operation.) In the previous formula, `bit_mask` is a bit mask created from individual bit flags that need to be ignored in the bit field.

Example

Table 1: Examples of Boolean Mask Computations (default parameters and 8-bit data type)

Bit Field	Bit Mask	~(Bit Mask)	Bit Field & ~(Bit Mask)	Boolean Mask
11011001 (217)	01010000 (80)	10101111 (175)	10001001 (137)	True
11011001 (217)	10101111 (175)	01010000 (80)	01010000 (80)	True
00001001 (9)	01001001 (73)	10110110 (182)	00000000 (0)	False
00001001 (9)	00000000 (0)	11111111 (255)	00001001 (9)	True
00001001 (9)	11111111 (255)	00000000 (0)	00000000 (0)	False

Specifying Bit Flags

`bitfield_to_boolean_mask` accepts either an integer bit mask or lists of bit flags. Lists of bit flags will be combined into a bit mask and can be provided either as a Python list of **integer bit flag values** or as a comma-separated (or `+`-separated) list of integer bit flag values. Consider the bit mask from the first example in Table 1. In this case `ignore_flags` can be set either to:

- An integer value bit mask 80
- A Python list indicating individual non-zero *bit flag values:* `[16, 64]`
- A string of comma-separated *bit flag values or mnemonic names*: `'16,64'`, `'CR,WARM'`
- A string of `+`-separated *bit flag values or mnemonic names*: `'16+64'`, `'CR+WARM'`

Example

To specify bit flags:

```
>>> from astropy.nddata import bitmask
>>> import numpy as np
>>> bitmask.bitfield_to_boolean_mask(217, ignore_flags=80)
array(True...)
>>> bitmask.bitfield_to_boolean_mask(217, ignore_flags='16,64')
array(True...)
>>> bitmask.bitfield_to_boolean_mask(217, ignore_flags=[16, 64])
array(True...)
>>> bitmask.bitfield_to_boolean_mask(9, ignore_flags=[1, 8, 64])
array(False...)
>>> bitmask.bitfield_to_boolean_mask([9, 10, 73, 217],
ignore_flags='1,8,64')
array([False,  True, False,  True]...)
```

It is also possible to specify the type of the output mask:

```
>>> bitmask.bitfield_to_boolean_mask([9, 10, 73, 217],
ignore_flags='1,8,64', dtype=np.uint8)
array([0, 1, 0, 1], dtype=uint8)
```

In order to use lists of mnemonic bit flags names, one must provide a map, a subclass of **`BitFlagNameMap`**, that can be used to map mnemonic names to bit flag values. Normally these maps should be provided by a third-party package supporting a specific instrument. Each bit flag in the map may also contain a string comment following the flag value. In the example below we define a simple mask map:

```
>>> from astropy.nddata.bitmask import BitFlagNameMap
>>> class ST_DQ(BitFlagNameMap):
...     CR = 1
...     CLOUDY = 4
...     RAINY = 8, 'Dome closed'
...     HOT = 32
...     DEAD = 64
```

```
>>> bitmask.bitfield_to_boolean_mask([9, 10, 73, 217],
ignore_flags='CR,RAINY,DEAD',
...                                     dtype=np.uint8,
flag_name_map=ST_DQ)
array([0, 1, 0, 1], dtype=uint8)
```

Using Bit Flags Name Maps

In order to allow the use of mnemonic bit flag names to describe the flags to be taken into consideration or ignored when creating a *boolean* mask, we use bit flag name maps. These maps perform case-insensitive translation of mnemonic bit flag names to the corresponding integer value.

Bit flag name maps are subclasses of **BitFlagNameMap** and can be constructed in two ways, either by directly subclassing **BitFlagNameMap**, e.g.,

```
>>> from astropy.nddata.bitmask import BitFlagNameMap
>>> class ST_DQ(BitFlagNameMap):
...     CR = 1
...     CLOUDY = 4
...     RAINY = 8
...
>>> class ST_CAM1_DQ(ST_DQ):
...     HOT = 16
...     DEAD = 32
```

or by using the **extend_bit_flag_map** class factory:

```
>>> from astropy.nddata.bitmask import extend_bit_flag_map
>>> ST_DQ = extend_bit_flag_map('ST_DQ', CR=1, CLOUDY=4, RAINY=8)
>>> ST_CAM1_DQ = extend_bit_flag_map('ST_CAM1_DQ', ST_DQ, HOT=16,
DEAD=32)
```

Note

Bit flag values must be integer numbers that are powers of 2.

Once constructed, bit flag values of a map cannot be modified, deleted, or added. Adding flags to a map is allowed only through subclassing using one of the two methods shown above or by adding lists of tuples of the form `('NAME', value)` to the class. This will create a new map class subclassed from the original map but containing the additional flags

```
>>> ST_CAM1_DQ = ST_DQ + [('HOT', 16), ('DEAD', 32)]
```

would result in an equivalent map as in the subclassing or class factory examples shown above.

Once a bit flag name map was created, the bit flag values can be accessed either as *case-insensitive* class attributes or keys in a dictionary:

```
>>> ST_CAM1_DQ.cloudy
4
>>> ST_CAM1_DQ['Rainy']
8
```

Modifying the Formula for Creating Boolean Masks

bitfield_to_boolean_mask provides several parameters that can be used to modify the formula used to create boolean masks.

Inverting Bit Masks

Sometimes it is more convenient to be able to specify those bit flags that *must be considered* when creating the boolean mask, and all other flags should be ignored.

Example

In **bitfield_to_boolean_mask** specifying bit flags that must be considered when creating the boolean mask can be accomplished by setting the parameter `flip_bits` to **True**. This effectively modifies equation (1) to:

```
(2)    boolean_mask = (bitfield & bit_mask) != 0
```

So, instead of:

```
>>> bitmask.bitfield_to_boolean_mask([9, 10, 73, 217],
ignore_flags=[1, 8, 64])
array([False,  True, False,  True]...)
```

You can obtain the same result as:

```
>>> bitmask.bitfield_to_boolean_mask(
...     [9, 10, 73, 217], ignore_flags=[2, 4, 16, 32, 128],
flip_bits=True
... )
array([False,  True, False,  True]...)
```

Note however, when `ignore_flags` is a comma-separated list of bit flag values, `flip_bits` cannot be set to either **True** or **False**. Instead, to flip bits of the bit mask formed from a string list of comma-separated bit flag values, you can prepend a single `~` to the list:

```
>>> bitmask.bitfield_to_boolean_mask([9, 10, 73, 217],
ignore_flags='~2+4+16+32+128')
array([False,  True, False,  True]...)
```

Inverting Boolean Masks

Other times, it may be more convenient to obtain an inverted mask in which flagged data are converted to `False` instead of `True`:

```
(3)    boolean_mask = (bitfield & ~bit_mask) == 0
```

This can be accomplished by changing the `good_mask_value` parameter from its default value (`False`) to `True`.

Example

To obtain an inverted mask in which flagged data are converted to `False` instead of `True`:

```
>>> bitmask.bitfield_to_boolean_mask([9, 10, 73, 217],
ignore_flags=[1, 8, 64],
...                                      good_mask_value=True)
array([ True, False,  True, False]...)
```

Decorating Functions to Accept NDData Objects

The `astropy.nddata` module includes a decorator `support_nddata()` that makes it convenient for developers and users to write functions that can accept `NDData` objects and also separate arguments.

Consider the following function:

```
def test(data, wcs=None, unit=None, n_iterations=3):
    ...
```

Now say that we want to be able to call the function as `test(nd)` where `nd` is an `NDData` instance. We can decorate this function using `support_nddata()`:

```
from astropy.nddata import support_nddata

@support_nddata
def test(data, wcs=None, unit=None, n_iterations=3):
    ...
```

Which makes it so that when the user calls `test(nd)`, the function would automatically be called with:

```
test(nd.data, wcs=nd.wcs, unit=nd.unit)
```

The decorator looks at the signature of the function and checks if any of the arguments are also properties of the `NDData` object, and passes them as individual arguments. The function can also be called with separate arguments as if it was not decorated.

A warning is emitted if an `NDData` property is set but the function does not accept it — for example, if `wcs` is set, but the function cannot support WCS objects. On the other hand, if an argument in the function does not exist in the `NDData` object or is not set, it is left to its default value.

If the function call succeeds, then the decorator returns the values from the function unmodified by default. However, in some cases we may want to return separate `data`, `wcs`, etc. if these were passed in separately, and a new `NDData` instance otherwise. To do this, you can specify `repack=True` in the decorator and provide a list of the names of the output arguments from the function:

```
@support_nddata(repack=True, returns=['data', 'wcs'])
def test(data, wcs=None, unit=None, n_iterations=3):
    ...
```

With this, the function will return separate values if `test` is called with separate arguments, and an object with the same class type as the input if the input is an `NDData` or subclass instance.

Finally, the decorator can be made to restrict input to specific `NDData` subclasses (and the subclasses of those) using the `accepts` option:

```
@support_nddata(accepts=CCDImage)
def test(data, wcs=None, unit=None, n_iterations=3):
    ...
```

NDData

Overview

`NDData` is based on `numpy.ndarray`-like `data` with additional meta attributes:

- `meta` for general metadata
- `unit` represents the physical unit of the data
- `uncertainty` for the uncertainty of the data
- `mask` indicates invalid points in the data
- `wcs` represents the relationship between the data grid and world

coordinates

Each of these attributes can be set during initialization or directly on the instance. Only the `data` cannot be directly set after creating the instance.

Data

The data is the base of **`NDData`** and is required to be **`numpy.ndarray`**-like. It is the only property that is required to create an instance and it cannot be directly set on the instance.

Example

To create an instance:

```
>>> import numpy as np
>>> from astropy.nddata import NDData
>>> array = np.array([[0, 1, 0], [1, 0, 1], [0, 1, 0]])
>>> ndd = NDData(array)
>>> ndd
NDData([[0, 1, 0],
        [1, 0, 1],
        [0, 1, 0]])
```

And access by the `data` attribute:

```
>>> ndd.data
array([[0, 1, 0],
       [1, 0, 1],
       [0, 1, 0]])
```

As already mentioned, it is not possible to set the data directly. So `ndd.data = np.arange(9)` will raise an exception. But the data can be modified in place:

```
>>> ndd.data[1,1] = 100
>>> ndd.data
array([[  0,   1,   0],
       [  1, 100,   1],
       [  0,   1,   0]])
```

Data During Initialization

During initialization it is possible to provide data that is not a **`numpy.ndarray`** but convertible to one.

Examples

To provide data that is convertible to a `numpy.ndarray`, you can pass a `list` containing numerical values:

```
>>> alist = [1, 2, 3, 4]
>>> ndd = NDData(alist)
>>> ndd.data  # data will be a numpy-array:
array([1, 2, 3, 4])
```

A nested `list` or `tuple` is possible, but if these contain non-numerical values the conversion might fail.

Besides input that is convertible to such an array, you can also use the `data` parameter to pass implicit additional information. For example, if the data is another `NDData` object it implicitly uses its properties:

```
>>> ndd = NDData(ndd, unit = 'm')
>>> ndd2 = NDData(ndd)
>>> ndd2.data  # It has the same data as ndd
array([1, 2, 3, 4])
>>> ndd2.unit  # but it also has the same unit as ndd
Unit("m")
```

Another possibility is to use a `Quantity` as a `data` parameter:

```
>>> import astropy.units as u
>>> quantity = np.ones(3) * u.cm  # this will create a Quantity
>>> ndd3 = NDData(quantity)
>>> ndd3.data
array([1., 1., 1.])
>>> ndd3.unit
Unit("cm")
```

Or a `numpy.ma.MaskedArray`:

```
>>> masked_array = np.ma.array([5,10,15], mask=[False, True, False])
>>> ndd4 = NDData(masked_array)
>>> ndd4.data
array([ 5, 10, 15])
>>> ndd4.mask
array([False,  True, False]...)
```

If such an implicitly passed property conflicts with an explicit parameter, the explicit parameter will be used and an info message will be issued:

```
>>> quantity = np.ones(3) * u.cm
>>> ndd6 = NDData(quantity, unit='m')
INFO: overwriting Quantity's current unit with specified unit.
[astropy.nddata.nddata]
>>> ndd6.data
array([1., 1., 1.])
>>> ndd6.unit
Unit("m")
```

The unit of the **`Quantity`** is being ignored and the unit is set to the explicitly passed one.

It might be possible to pass other classes as a `data` parameter as long as they have the properties `shape`, `dtype`, `__getitem__`, and `__array__`.

The purpose of this mechanism is to allow considerable flexibility in the objects used to store the data while providing a useful default (`numpy` array).

Mask

The `mask` is being used to indicate if data points are valid or invalid. **`NDData`** does not restrict this mask in any way but it is expected to follow the **`numpy.ma.MaskedArray`** convention in that the mask:

- Returns `True` for data points that are considered **invalid**.
- Returns `False` for those points that are **valid**.

Examples

One possibility is to create a mask by using `numpy`'s comparison operators:

```
>>> array = np.array([0, 1, 4, 0, 2])

>>> mask = array == 0  # Mask points containing 0
>>> mask
array([ True, False, False,  True, False]...)

>>> other_mask = array > 1  # Mask points with a value greater than 1
>>> other_mask
array([False, False,  True, False,  True]...)
```

And initialize the **`NDData`** instance using the `mask` parameter:

```
>>> ndd = NDData(array, mask=mask)
>>> ndd.mask
array([ True, False, False,  True, False]...)
```

Or by replacing the mask:

```
>>> ndd.mask = other_mask
>>> ndd.mask
array([False, False,  True, False,  True]...)
```

There is no requirement that the mask actually be a `numpy` array; for example, a function which evaluates a mask value as needed is acceptable as long as it follows the convention that `True` indicates a value that should be ignored.

Unit

The `unit` represents the unit of the data values. It is required to be **`Unit`**-like or a string that can be converted to such a **`Unit`**:

```
>>> import astropy.units as u
>>> ndd = NDData([1, 2, 3, 4], unit="meter")  # using a string
>>> ndd.unit
Unit("m")
```

..note::

Setting the `unit` on an instance is not possible.

Uncertainties

The `uncertainty` represents an arbitrary representation of the error of the data values. To indicate which kind of uncertainty representation is used, the `uncertainty` should have an `uncertainty_type` property. If no such property is found it will be wrapped inside a **`UnknownUncertainty`**.

The `uncertainty_type` should follow the **`StdDevUncertainty`** convention in that it returns a short string like `"std"` for an uncertainty given in standard deviation. Other examples are **`VarianceUncertainty`** and **`InverseVariance`**.

Examples

Like the other properties the `uncertainty` can be set during initialization:

```
>>> from astropy.nddata import StdDevUncertainty
>>> array = np.array([10, 7, 12, 22])
>>> uncert = StdDevUncertainty(np.sqrt(array))
>>> ndd = NDData(array, uncertainty=uncert)
>>> ndd.uncertainty
StdDevUncertainty([3.16227766, 2.64575131, 3.46410162, 4.69041576])
```

Or on the instance directly:

```
>>> other_uncert = StdDevUncertainty([2,2,2,2])
>>> ndd.uncertainty = other_uncert
>>> ndd.uncertainty
StdDevUncertainty([2, 2, 2, 2])
```

But it will print an info message if there is no `uncertainty_type`:

```
>>> ndd.uncertainty = np.array([5, 1, 2, 10])
INFO: uncertainty should have attribute uncertainty_type.
[astropy.nddata.nddata]
>>> ndd.uncertainty
UnknownUncertainty([ 5,  1,  2, 10])
```

WCS

The `wcs` should contain a mapping from the gridded data to world coordinates. There are no restrictions placed on the property currently but it may be restricted to an **`WCS`** object or a more generalized WCS object in the future.

Note

Like the unit the `wcs` cannot be set on an instance.

Metadata

The `meta` property contains all further meta information that does not fit any other property.

Examples

If the `meta` property is given it must be **`dict`**-like:

```
>>> ndd = NDData([1,2,3], meta={'observer': 'myself'})
>>> ndd.meta
{'observer': 'myself'}
```

dict-like means it must be a mapping from some keys to some values. This also includes Header objects:

```
>>> from astropy.io import fits
>>> header = fits.Header()
>>> header['observer'] = 'Edwin Hubble'
>>> ndd = NDData(np.zeros([10, 10]), meta=header)
>>> ndd.meta['observer']
'Edwin Hubble'
```

If the `meta` property is not provided or explicitly set to `None`, it will default to an empty collections.OrderedDict:

```
>>> ndd.meta = None
>>> ndd.meta
OrderedDict()

>>> ndd = NDData([1,2,3])
>>> ndd.meta
OrderedDict()
```

The `meta` object therefore supports adding or updating these values:

```
>>> ndd.meta['exposure_time'] = 340.
>>> ndd.meta['filter'] = 'J'
```

Elements of the metadata dictionary can be set to any valid Python object:

```
>>> ndd.meta['history'] = ['calibrated', 'aligned', 'flat-fielded']
```

Initialization with Copy

The default way to create an NDData instance is to try saving the parameters as references to the original rather than as copy. Sometimes this is not possible because the internal mechanics do not allow for this.

Examples

If the `data` is a list then during initialization this is copied while converting to a ndarray. But it is also possible to enforce copies during initialization by setting the `copy` parameter to `True`:

```
>>> array = np.array([1, 2, 3, 4])
>>> ndd = NDData(array)
>>> ndd.data[2] = 10
>>> array[2]  # Original array has changed
10

>>> ndd2 = NDData(array, copy=True)
>>> ndd2.data[2] = 3
>>> array[2]  # Original array hasn't changed.
10
```

Note

In some cases setting `copy=True` will copy the `data` twice. Known cases are if the `data` is a **`list`** or **`tuple`**.

Converting NDData to Other Classes

There is limited support to convert a **`NDData`** instance to other classes. In the process some properties might be lost.

```
>>> data = np.array([1, 2, 3, 4])
>>> mask = np.array([True, False, False, True])
>>> unit = 'm'
>>> ndd = NDData(data, mask=mask, unit=unit)
```

`numpy.ndarray`

Converting the `data` to an array:

```
>>> array = np.asarray(ndd.data)
>>> array
array([1, 2, 3, 4])
```

Though using `np.asarray` is not required, in most cases it will ensure that the result is always a **`numpy.ndarray`**

`numpy.ma.MaskedArray`

Converting the `data` and `mask` to a MaskedArray:

```
>>> masked_array = np.ma.array(ndd.data, mask=ndd.mask)
>>> masked_array
masked_array(data=[--, 2, 3, --],
             mask=[ True, False, False,  True],
       fill_value=999999)
```

Quantity

Converting the `data` and `unit` to a Quantity:

```
>>> quantity = u.Quantity(ndd.data, unit=ndd.unit)
>>> quantity
<Quantity [1., 2., 3., 4.] m>
```

Note

Ideally, you would construct masked quantities, but these are not properly supported: many operations on them fail.

Mixins for Added Functionality

Slicing and Indexing NDData

Introduction

This page only deals with peculiarities that apply to **NDData**-like classes. For a tutorial about slicing/indexing see the python documentation and numpy documentation.

Warning

NDData and **NDDataRef** enforce almost no restrictions on the properties, so it might happen that some **valid but unusual** combinations of properties always result in an IndexError or incorrect results. In this case, see Subclassing on how to customize slicing for a particular property.

Slicing NDDataRef

Unlike **NDData** the class **NDDataRef** implements slicing or indexing. The result will be wrapped inside the same class as the sliced object.

Getting one element:

```
>>> import numpy as np
>>> from astropy.nddata import NDDataRef

>>> data = np.array([1, 2, 3, 4])
>>> ndd = NDDataRef(data)
>>> ndd[1]
NDDataRef(2)
```

Getting a sliced portion of the original:

```
>>> ndd[1:3]  # Get element 1 (inclusive) to 3 (exclusive)
NDDataRef([2, 3])
```

This will return a reference (and as such **not a copy**) of the original properties, so changing a slice will affect the original:

```
>>> ndd_sliced = ndd[1:3]
>>> ndd_sliced.data[0] = 5
>>> ndd_sliced
NDDataRef([5, 3])
>>> ndd
NDDataRef([1, 5, 3, 4])
```

But only the one element that was indexed is affected (for example, `ndd_sliced = ndd[1]`). The element is a scalar and changes will not propagate to the original.

Slicing NDDataRef Including Attributes

In the case that a `mask`, or `uncertainty` is present, this attribute will be sliced too:

```
>>> from astropy.nddata import StdDevUncertainty
>>> data = np.array([1, 2, 3, 4])
>>> mask = data > 2
>>> uncertainty = StdDevUncertainty(np.sqrt(data))
>>> ndd = NDDataRef(data, mask=mask, uncertainty=uncertainty)
>>> ndd_sliced = ndd[1:3]

>>> ndd_sliced.data
array([2, 3])

>>> ndd_sliced.mask
array([False,  True]...)

>>> ndd_sliced.uncertainty
StdDevUncertainty([1.41421356, 1.73205081])
```

`unit` and `meta`, however, will be unaffected.

If any of the attributes are set but do not implement slicing, an info will be printed and the property will be kept as is:

```
>>> data = np.array([1, 2, 3, 4])
>>> mask = False
>>> uncertainty = StdDevUncertainty(0)
>>> ndd = NDDataRef(data, mask=mask, uncertainty=uncertainty)
>>> ndd_sliced = ndd[1:3]
INFO: uncertainty cannot be sliced. [astropy.nddata.mixins.ndslicing]
INFO: mask cannot be sliced. [astropy.nddata.mixins.ndslicing]

>>> ndd_sliced.mask
False
```

Slicing NDData with World Coordinates

If `wcs` is set, it must be either implement **BaseLowLevelWCS** or **BaseHighLevelWCS**. This means that only integer or range slices without a step are supported. So slices like `[::10]` or array or boolean based slices will not work.

If you want to slice an `NDData` object called `ndd` without the WCS you can remove the WCS from the `NDData` object by running:

```
>>> ndd.wcs = None
```

Removing Masked Data

Warning

If `wcs` is set this will **NOT** be possible. But you can work around this by setting the wcs attribute to **None** with `ndd.wcs = None` before slicing.

By convention, the `mask` attribute indicates if a point is valid or invalid. So we are able to get all valid data points by slicing with the mask.

Examples

To get all of the valid data points by slicing with the mask:

```
>>> data = np.array([[1,2,3],[4,5,6],[7,8,9]])
>>> mask = np.array([[0,1,0],[1,1,1],[0,0,1]], dtype=bool)
>>> uncertainty = StdDevUncertainty(np.sqrt(data))
>>> ndd = NDDataRef(data, mask=mask, uncertainty=uncertainty)
>>> # don't forget that ~ or you'll get the invalid points
>>> ndd_sliced = ndd[~ndd.mask]
>>> ndd_sliced
NDDataRef([1, 3, 7, 8])

>>> ndd_sliced.mask
array([False, False, False, False]...)
```

```
>>> ndd_sliced.uncertainty
StdDevUncertainty([1.        , 1.73205081, 2.64575131, 2.82842712])
```

Or all invalid points:

```
>>> ndd_sliced = ndd[ndd.mask] # without the ~ now!
>>> ndd_sliced
NDDataRef([2, 4, 5, 6, 9])

>>> ndd_sliced.mask
array([ True,  True,  True,  True,  True]...)

>>> ndd_sliced.uncertainty
StdDevUncertainty([1.41421356, 2.        , 2.23606798, 2.44948974, 3.
])
```

Note

The result of this kind of indexing (boolean indexing) will always be one-dimensional!

NDData Arithmetic

Introduction

`NDDataRef` implements the following arithmetic operations:

- Addition: `add()`
- Subtraction: `subtract()`
- Multiplication: `multiply()`
- Division: `divide()`

Using Basic Arithmetic Methods

Using the standard arithmetic methods requires that the first operand is an `NDDataRef` instance:

```
>>> from astropy.nddata import NDDataRef
>>> from astropy.wcs import WCS
>>> import numpy as np
>>> ndd1 = NDDataRef([1, 2, 3, 4])
```

While the requirement for the second operand is that it must be convertible to the first operand. It can be a number:

```
>>> ndd1.add(3)
NDDataRef([4, 5, 6, 7])
```

Or a **list**:

```
>>> ndd1.subtract([1,1,1,1])
NDDataRef([0, 1, 2, 3])
```

Or a **numpy.ndarray**:

```
>>> ndd1.multiply(np.arange(4, 8))
NDDataRef([ 4, 10, 18, 28])
>>> ndd1.divide(np.arange(1,13).reshape(3,4))  # a 3 x 4 numpy array
NDDataRef([[1.        , 1.        , 1.        , 1.        ],
           [0.2       , 0.33333333, 0.42857143, 0.5       ],
           [0.11111111, 0.2       , 0.27272727, 0.33333333]])
```

Here, broadcasting takes care of the different dimensions. Several other classes are also possible.

Using Arithmetic Classmethods

Here both operands do not need to be **NDDataRef**-like:

```
>>> NDDataRef.add(1, 3)
NDDataRef(4)
```

To wrap the result of an arithmetic operation between two Quantities:

```
>>> import astropy.units as u
>>> ndd = NDDataRef.multiply([1,2] * u.m, [10, 20] * u.cm)
>>> ndd
NDDataRef([10., 40.])
>>> ndd.unit
Unit("cm m")
```

Or take the inverse of an **NDDataRef** object:

```
>>> NDDataRef.divide(1, ndd1)
NDDataRef([1.        , 0.5       , 0.33333333, 0.25      ])
```

Possible Operands

The possible types of input for operands are:

- Scalars of any type
- Lists containing numbers (or nested lists)

- `numpy` arrays
- `numpy` masked arrays
- `astropy` quantities
- Other `nddata` classes or subclasses

Advanced Options

The normal Python operators `+`, `-`, etc. are not implemented because the methods provide several options on how to proceed with the additional attributes.

Data and Unit

For `data` and `unit` there are no parameters. Every arithmetic operation lets the **`astropy.units.Quantity`**-framework evaluate the result or fail and abort the operation.

Adding two **`NDData`** objects with the same unit works:

```
>>> ndd1 = NDDataRef([1,2,3,4,5], unit='m')
>>> ndd2 = NDDataRef([100,150,200,50,500], unit='m')

>>> ndd = ndd1.add(ndd2)
>>> ndd.data
array([101., 152., 203.,  54., 505.])
>>> ndd.unit
Unit("m")
```

Adding two **`NDData`** objects with compatible units also works:

```
>>> ndd1 = NDDataRef(ndd1, unit='pc')
INFO: overwriting NDData's current unit with specified unit.
[astropy.nddata.nddata]
>>> ndd2 = NDDataRef(ndd2, unit='lyr')
INFO: overwriting NDData's current unit with specified unit.
[astropy.nddata.nddata]

>>> ndd = ndd1.subtract(ndd2)
>>> ndd.data
array([ -29.66013938,  -43.99020907,  -58.32027876,  -11.33006969,
       -148.30069689])
>>> ndd.unit
Unit("pc")
```

This will keep by default the unit of the first operand. However, units will not be decomposed during division:

```
>>> ndd = ndd2.divide(ndd1)
>>> ndd.data
array([100.        ,  75.        ,  66.66666667,  12.5       , 100.
])
>>> ndd.unit
Unit("lyr / pc")
```

Mask

The `handle_mask` parameter for the arithmetic operations implements what the resulting mask will be. There are several options.

- `None`, the result will have no `mask`:

```
>>> ndd1 = NDDataRef(1, mask=True)
>>> ndd2 = NDDataRef(1, mask=False)
>>> ndd1.add(ndd2, handle_mask=None).mask is None
True
```

- `"first_found"` or `"ff"`, the result will have the `mask` of the first operand or if that is `None`, the `mask` of the second operand:

```
>>> ndd1 = NDDataRef(1, mask=True)
>>> ndd2 = NDDataRef(1, mask=False)
>>> ndd1.add(ndd2, handle_mask="first_found").mask
True
>>> ndd3 = NDDataRef(1)
>>> ndd3.add(ndd2, handle_mask="first_found").mask
False
```

- A function (or an arbitrary callable) that takes at least two arguments. For example, **`numpy.logical_or`** is the default:

```
>>> ndd1 = NDDataRef(1, mask=np.array([True, False, True, False]))
>>> ndd2 = NDDataRef(1, mask=np.array([True, False, False, True]))
>>> ndd1.add(ndd2).mask
array([ True, False,  True,  True]...)
```

  This defaults to `"first_found"` in case only one `mask` is not None:

```
>>> ndd1 = NDDataRef(1)
>>> ndd2 = NDDataRef(1, mask=np.array([True, False, False, True]))
>>> ndd1.add(ndd2).mask
array([ True, False, False,  True]...)
```

  Custom functions are also possible:

```
>>> def take_alternating_values(mask1, mask2, start=0):
...     result = np.zeros(mask1.shape, dtype=np.bool_)
...     result[start::2] = mask1[start::2]
...     result[start+1::2] = mask2[start+1::2]
...     return result
```

This function is nonsense, but we can still see how it performs:

```
>>> ndd1 = NDDataRef(1, mask=np.array([True, False, True, False]))
>>> ndd2 = NDDataRef(1, mask=np.array([True, False, False, True]))
>>> ndd1.add(ndd2, handle_mask=take_alternating_values).mask
array([ True, False,  True,  True]...)
```

Additional parameters can be given by prefixing them with `mask_` (which will be stripped before passing it to the function):

```
>>> ndd1.add(ndd2, handle_mask=take_alternating_values,
mask_start=1).mask
array([False, False, False, False]...)
>>> ndd1.add(ndd2, handle_mask=take_alternating_values,
mask_start=2).mask
array([False, False,  True,  True]...)
```

Meta

The `handle_meta` parameter for the arithmetic operations implements what the resulting `meta` will be. The options are the same as for the `mask`:

- If `None` the resulting `meta` will be an empty **`collections.OrderedDict`**.

```
>>> ndd1 = NDDataRef(1, meta={'object': 'sun'})
>>> ndd2 = NDDataRef(1, meta={'object': 'moon'})
>>> ndd1.add(ndd2, handle_meta=None).meta
OrderedDict()
```

For `meta` this is the default so you do not need to pass it in this case:

```
>>> ndd1.add(ndd2).meta
OrderedDict()
```

- If `"first_found"` or `"ff"`, the resulting `meta` will be the `meta` of the first operand or if that contains no keys, the `meta` of the second operand is taken.

```
>>> ndd1 = NDDataRef(1, meta={'object': 'sun'})
>>> ndd2 = NDDataRef(1, meta={'object': 'moon'})
>>> ndd1.add(ndd2, handle_meta='ff').meta
{'object': 'sun'}
```

- If it is a `callable` it must take at least two arguments. Both `meta` attributes will be passed to this function (even if one or both of them are empty) and the callable evaluates the result's `meta`. For example, a function that merges these two:

```
>>> # It's expected with arithmetics that the result is not a
reference,
>>> # so we need to copy
>>> from copy import deepcopy

>>> def combine_meta(meta1, meta2):
...     if not meta1:
...         return deepcopy(meta2)
...     elif not meta2:
...         return deepcopy(meta1)
...     else:
...         meta_final = deepcopy(meta1)
...         meta_final.update(meta2)
...         return meta_final

>>> ndd1 = NDDataRef(1, meta={'time': 'today'})
>>> ndd2 = NDDataRef(1, meta={'object': 'moon'})
>>> ndd1.subtract(ndd2, handle_meta=combine_meta).meta
{'object': 'moon', 'time': 'today'}
```

  Here again additional arguments for the function can be passed in using the prefix `meta_` (which will be stripped away before passing it to this function). See the description for the mask-attribute for further details.

World Coordinate System (WCS)

The `compare_wcs` argument will determine what the result's `wcs` will be or if the operation should be forbidden. The possible values are identical to `mask` and `meta`:

- If `None` the resulting `wcs` will be an empty `None`.

```
>>> ndd1 = NDDataRef(1, wcs=None)
>>> ndd2 = NDDataRef(1, wcs=WCS())
>>> ndd1.add(ndd2, compare_wcs=None).wcs is None
True
```

- If `"first_found"` or `"ff"` the resulting `wcs` will be the `wcs` of the first operand or if that is `None`, the `meta` of the second operand is taken.

```
>>> wcs = WCS()
>>> ndd1 = NDDataRef(1, wcs=wcs)
>>> ndd2 = NDDataRef(1, wcs=None)
>>> str(ndd1.add(ndd2, compare_wcs='ff').wcs) == str(wcs)
True
```

- If it is a `callable` it must take at least two arguments. Both `wcs` attributes will be passed to this function (even if one or both of them are `None`) and the callable should return `True` if these `wcs` are identical (enough) to allow the arithmetic operation or `False` if the arithmetic operation should be aborted with a `ValueError`. If `True` the `wcs` are identical and the first one is used for the result:

```
>>> def compare_wcs_scalar(wcs1, wcs2, allowed_deviation=0.1):
...     if wcs1 is None and wcs2 is None:
...         return True  # both have no WCS so they are identical
...     if wcs1 is None or wcs2 is None:
...         return False  # one has WCS, the other doesn't not
possible
...     else:
...         # Consider wcs close if centers are close enough
...         return all(abs(wcs1.wcs.crpix - wcs2.wcs.crpix) <
allowed_deviation)

>>> ndd1 = NDDataRef(1, wcs=None)
>>> ndd2 = NDDataRef(1, wcs=None)
>>> ndd1.subtract(ndd2, compare_wcs=compare_wcs_scalar).wcs
```

Additional arguments can be passed in prefixing them with `wcs_` (this prefix will be stripped away before passing it to the function):

```
>>> ndd1 = NDDataRef(1, wcs=WCS())
>>> ndd1.wcs.wcs.crpix = [1, 1]
>>> ndd2 = NDDataRef(1, wcs=WCS())
>>> ndd1.subtract(ndd2, compare_wcs=compare_wcs_scalar,
wcs_allowed_deviation=2).wcs.wcs.crpix
array([1., 1.])
```

If you are using **WCS** objects, a very handy function to use might be:

```
>>> def wcs_compare(wcs1, wcs2, *args, **kwargs):
...     return wcs1.wcs.compare(wcs2.wcs, *args, **kwargs)
```

See **astropy.wcs.Wcsprm.compare()** for the arguments this comparison allows.

Uncertainty

The `propagate_uncertainties` argument can be used to turn the propagation of uncertainties on or off.

- If `None` the result will have no uncertainty:

```
>>> from astropy.nddata import StdDevUncertainty
>>> ndd1 = NDDataRef(1, uncertainty=StdDevUncertainty(0))
>>> ndd2 = NDDataRef(1, uncertainty=StdDevUncertainty(1))
>>> ndd1.add(ndd2, propagate_uncertainties=None).uncertainty is
None
True
```

- If `False` the result will have the first found uncertainty.

Note

Setting `propagate_uncertainties=False` is generally not recommended.

- If `True` both uncertainties must be `NDUncertainty` subclasses that implement propagation. This is possible for **StdDevUncertainty**:

```
>>> ndd1 = NDDataRef(1, uncertainty=StdDevUncertainty([10]))
>>> ndd2 = NDDataRef(1, uncertainty=StdDevUncertainty([10]))
>>> ndd1.add(ndd2, propagate_uncertainties=True).uncertainty
StdDevUncertainty([14.14213562])
```

Uncertainty with Correlation

If `propagate_uncertainties` is `True` you can also give an argument for `uncertainty_correlation`. **StdDevUncertainty** cannot keep track of its correlations by itself, but it can evaluate the correct resulting uncertainty if the correct `correlation` is given.

The default (`0`) represents uncorrelated while `1` means correlated and `-1` anti-correlated. If given a **numpy.ndarray** it should represent the element-wise correlation coefficient.

Examples

Without correlation, subtracting an **NDDataRef** instance from itself results in a non-zero uncertainty:

```
>>> ndd1 = NDDataRef(1, uncertainty=StdDevUncertainty([10]))
>>> ndd1.subtract(ndd1, propagate_uncertainties=True).uncertainty
```

```
StdDevUncertainty([14.14213562])
```

Given a correlation of `1` (because they clearly correlate) gives the correct uncertainty of `0`:

```
>>> ndd1 = NDDataRef(1, uncertainty=StdDevUncertainty([10]))
>>> ndd1.subtract(ndd1, propagate_uncertainties=True,
...                uncertainty_correlation=1).uncertainty
StdDevUncertainty([0.])
```

Which would be consistent with the equivalent operation `ndd1 * 0`:

```
>>> ndd1.multiply(0, propagate_uncertainties=True).uncertainty
StdDevUncertainty([0.])
```

Warning

The user needs to calculate or know the appropriate value or array manually and pass it to `uncertainty_correlation`. The implementation follows general first order error propagation formulas. See, for example: Wikipedia.

You can also give element-wise correlations:

```
>>> ndd1 = NDDataRef([1,1,1,1],
uncertainty=StdDevUncertainty([1,1,1,1]))
>>> ndd2 = NDDataRef([2,2,2,2],
uncertainty=StdDevUncertainty([2,2,2,2]))
>>> ndd1.add(ndd2,uncertainty_correlation=np.array([1,0.5,0,-
1])).uncertainty
StdDevUncertainty([3.        , 2.64575131, 2.23606798, 1.        ])
```

The correlation `np.array([1, 0.5, 0, -1])` would indicate that the first element is fully correlated and the second element partially correlates, while the third element is uncorrelated, and the fourth is anti-correlated.

Uncertainty with Unit

`StdDevUncertainty` implements correct error propagation even if the unit of the data differs from the unit of the uncertainty:

```
>>> ndd1 = NDDataRef([10], unit='m',
uncertainty=StdDevUncertainty([10], unit='cm'))
>>> ndd2 = NDDataRef([20], unit='m',
uncertainty=StdDevUncertainty([10]))
>>> ndd1.subtract(ndd2, propagate_uncertainties=True).uncertainty
StdDevUncertainty([10.00049999])
```

But it needs to be convertible to the unit for the data.

I/O Mixin

The I/O mixin, **NDIOMixin**, adds `read` and `write` methods that use the `astropy` I/O registry.

The mixin itself creates the read/write methods; it does not register any readers or writers with the I/O registry. Subclasses of **NDDataBase** or **NDData** need to include this mixin, implement a reader and writer, *and* register it with the I/O framework. See I/O Registry (astropy.io.registry) for details.

Subclassing

NDData

This class serves as the base for subclasses that use a **numpy.ndarray** (or something that presents a `numpy`-like interface) as the `data` attribute.

Note

Each attribute is saved as an attribute with one leading underscore. For example, the `data` is saved as `_data` and the `mask` as `_mask`, and so on.

Adding Another Property

```
>>> from astropy.nddata import NDData
```

```
>>> class NDDataWithFlags(NDData):
...     def __init__(self, *args, **kwargs):
...         # Remove flags attribute if given and pass it to the
setter.
...         self.flags = kwargs.pop('flags') if 'flags' in kwargs
else None
...         super().__init__(*args, **kwargs)
...
...     @property
...     def flags(self):
...         return self._flags
...
...     @flags.setter
...     def flags(self, value):
...         self._flags = value
```

```
>>> ndd = NDDataWithFlags([1,2,3])
>>> ndd.flags is None
True
```

```
>>> ndd = NDDataWithFlags([1,2,3], flags=[0, 0.2, 0.3])
>>> ndd.flags
[0, 0.2, 0.3]
```

Note

To simplify subclassing, each setter (except for `data`) is called during `__init__` so putting restrictions on any attribute can be done inside the setter and will also apply during instance creation.

Customize the Setter for a Property

```
>>> import numpy as np
```

```
>>> class NDDataMaskBoolNumpy(NDData):
...
...     @NDData.mask.setter
...     def mask(self, value):
...         # Convert mask to boolean numpy array.
...         self._mask = np.array(value, dtype=np.bool_)
```

```
>>> ndd = NDDataMaskBoolNumpy([1,2,3])
>>> ndd.mask = [True, False, True]
>>> ndd.mask
array([ True, False,  True]...)
```

Extend the Setter for a Property

`unit`, `meta`, and `uncertainty` implement some additional logic in their setter so subclasses might define a call to the superclass and let the super property set the attribute afterwards:

```
>>> import numpy as np

>>> class NDDataUncertaintyShapeChecker(NDData):
...
...     @NDData.uncertainty.setter
...     def uncertainty(self, value):
```

```
...         value = np.asarray(value)
...         if value.shape != self.data.shape:
...             raise ValueError('uncertainty must have the same
shape as the data.')
...         # Call the setter of the super class in case it might
contain some
...         # important logic (only True for meta, unit and
uncertainty)
...         super(NDDataUncertaintyShapeChecker,
self.__class__).uncertainty.fset(self, value)
...         # Unlike "super(cls_name, cls_name).uncertainty.fset" or
...         # or "NDData.uncertainty.fset" this will respect Pythons
method
...         # resolution order.

>>> ndd = NDDataUncertaintyShapeChecker([1,2,3], uncertainty=[2,3,4])
INFO: uncertainty should have attribute uncertainty_type.
[astropy.nddata.nddata]
>>> ndd.uncertainty
UnknownUncertainty([2, 3, 4])
```

Having a Setter for the Data

```
>>> class NDDataWithDataSetter(NDData):
...
...     @NDData.data.setter
...     def data(self, value):
...         self._data = np.asarray(value)
```

```
>>> ndd = NDDataWithDataSetter([1,2,3])
>>> ndd.data = [3,2,1]
>>> ndd.data
array([3, 2, 1])
```

NDDataRef

NDDataRef itself inherits from **NDData** so any of the possibilities there also apply to NDDataRef. But NDDataRef also inherits from the Mixins:

- **NDSlicingMixin**
- **NDArithmeticMixin**
- **NDIOMixin**

Which allow additional operations.

Add Another Arithmetic Operation

Adding another operation is possible provided the `data` and `unit` allow it within the framework of **`Quantity`**.

Examples

To add a power function:

```
>>> from astropy.nddata import NDDataRef
>>> import numpy as np
>>> from astropy.utils import sharedmethod

>>> class NDDataPower(NDDataRef):
...     @sharedmethod # sharedmethod to allow it also as classmethod
...     def pow(self, operand, operand2=None, **kwargs):
...         # the uncertainty doesn't allow propagation so set it to
None
...         kwargs['propagate_uncertainties'] = None
...         # Call the _prepare_then_do_arithmetic function with the
...         # numpy.power ufunc.
...         return self._prepare_then_do_arithmetic(np.power,
operand,
...                                                     operand2,
**kwargs)
```

This can be used like the other arithmetic methods similar to **`add()`**. So it works when calling it on the class or the instance:

```
>>> ndd = NDDataPower([1,2,3])

>>> # using it on the instance with one operand
>>> ndd.pow(3)
NDDataPower([ 1,  8, 27])

>>> # using it on the instance with two operands
>>> ndd.pow([1,2,3], [3,4,5])
NDDataPower([  1,  16, 243])

>>> # or using it as classmethod
>>> NDDataPower.pow(6, [1,2,3])
NDDataPower([  6,  36, 216])
```

To allow propagation also with `uncertainty` see subclassing **`NDUncertainty`**.

The `_prepare_then_do_arithmetic` implements the relevant checks if it was called on the class or the instance, and if one or two operands were given, converts the operands, if necessary, to the appropriate classes. Overriding `_prepare_then_do_arithmetic` in subclasses should be avoided if possible.

Arithmetic on an Existing Property

Customizing how an existing property is handled during arithmetic is possible with some arguments to the function calls such as **add()**, but it is possible to hardcode behavior too. The actual operation on the attribute (except for `unit`) is done in a method `_arithmetic_*` where `*` is the name of the property.

Examples

To customize how the `meta` will be affected during arithmetics:

```
>>> from astropy.nddata import NDDataRef

>>> from copy import deepcopy
>>> class NDDataWithMetaArithmetics(NDDataRef):
...
...     def _arithmetic_meta(self, operation, operand, handle_mask,
**kwds):
...         # the function must take the arguments:
...         # operation (numpy-ufunc like np.add, np.subtract, ...)
...         # operand (the other NDData-like object, already wrapped
as NDData)
...         # handle_mask (see description for "add")
...
...         # The meta is dict like but we want the keywords exposure
to change
...         # Anticipate that one or both might have no meta and take
the first one that has
...         result_meta = deepcopy(self.meta) if self.meta else
deepcopy(operand.meta)
...         # Do the operation on the keyword if the keyword exists
...         if result_meta and 'exposure' in result_meta:
...             result_meta['exposure'] =
operation(result_meta['exposure'], operand.data)
...         return result_meta # return it
```

To trigger this method, the `handle_meta` argument to arithmetic methods can be anything except `None` or `"first_found"`:

```
>>> ndd = NDDataWithMetaArithmetics([1,2,3], meta={'exposure': 10})
>>> ndd2 = ndd.add(10, handle_meta='')
>>> ndd2.meta
{'exposure': 20}

>>> ndd3 = ndd.multiply(0.5, handle_meta='')
>>> ndd3.meta
```

```
{'exposure': 5.0}
```

Warning

To use these internal `_arithmetic_*` methods there are some restrictions on the attributes when calling the operation:

- `mask`: `handle_mask` must not be `None`, `"ff"`, or `"first_found"`.
- `wcs`: `compare_wcs` argument with the same restrictions as mask.
- `meta`: `handle_meta` argument with the same restrictions as mask.
- `uncertainty`: `propagate_uncertainties` must be `None` or evaluate to `False`. `arithmetic_uncertainty` must also accept different arguments: `operation`, `operand`, `result`, `correlation`, `**kwargs`.

Changing the Default Argument for Arithmetic Operations

If the goal is to change the default value of an existing parameter for arithmetic methods, such as when explicitly specifying the parameter each time you call an arithmetic operation is too much effort, you can change the default value of existing parameters by changing it in the method signature of `_arithmetic`.

Example

To change the default value of an existing parameter for arithmetic methods:

```
>>> from astropy.nddata import NDDataRef
>>> import numpy as np

>>> class NDDDiffAritDefaults(NDDataRef):
...     def _arithmetic(self, *args, **kwargs):
...         # Changing the default of handle_mask to None
...         if 'handle_mask' not in kwargs:
...             kwargs['handle_mask'] = None
...         # Call the original with the updated kwargs
...         return super()._arithmetic(*args, **kwargs)

>>> ndd1 = NDDDiffAritDefaults(1, mask=False)
>>> ndd2 = NDDDiffAritDefaults(1, mask=True)
>>> ndd1.add(ndd2).mask is None  # it will be None
True

>>> # But giving other values is still possible:
>>> ndd1.add(ndd2, handle_mask=np.logical_or).mask
```

```
True

>>> ndd1.add(ndd2, handle_mask="ff").mask
False
```

The parameter controlling how properties are handled are all keyword-only so using the `*args`, `**kwargs` approach allows you to only alter one default without needing to care about the positional order of arguments.

Arithmetic with an Additional Property

This also requires overriding the `_arithmetic` method. Suppose we have a `flags` attribute again:

```
>>> from copy import deepcopy
>>> import numpy as np

>>> class NDDataWithFlags(NDDataRef):
...     def __init__(self, *args, **kwargs):
...         # Remove flags attribute if given and pass it to the
setter.
...         self.flags = kwargs.pop('flags') if 'flags' in kwargs
else None
...         super().__init__(*args, **kwargs)
...
...     @property
...     def flags(self):
...         return self._flags
...
...     @flags.setter
...     def flags(self, value):
...         self._flags = value
...
...     def _arithmetic(self, operation, operand, *args, **kwargs):
...         # take all args and kwargs to allow arithmetic on the
other properties
...         # to work like before.
...
...         # do the arithmetics on the flags (pop the relevant
kwargs, if any!!!)
...         if self.flags is not None and operand.flags is not None:
...             result_flags = np.logical_or(self.flags,
operand.flags)
...             # np.logical_or is just a suggestion you can do what
you want
...         else:
...             if self.flags is not None:
...                 result_flags = deepcopy(self.flags)
...                 else:
...                     result_flags = deepcopy(operand.flags)
...
...             # Let the superclass do all the other attributes note
that
...             # this returns the result and a dictionary containing
other attributes
...             result, kwargs = super()._arithmetic(operation, operand,
*args, **kwargs)
...             # The arguments for creating a new instance are saved in
kwargs
...             # so we need to add another keyword "flags" and add the
processed flags
...             kwargs['flags'] = result_flags
...             return result, kwargs # these must be returned

>>> ndd1 = NDDataWithFlags([1,2,3], flags=np.array([1,0,1],
dtype=bool))
>>> ndd2 = NDDataWithFlags([1,2,3], flags=np.array([0,0,1],
dtype=bool))
>>> ndd3 = ndd1.add(ndd2)
>>> ndd3.flags
array([ True, False,  True]...)
```

Slicing an Existing Property

Suppose you have a class expecting a 2D `data` but the mask is only 1D. This would lead to problems if you were to slice in two dimensions.

```
>>> from astropy.nddata import NDDataRef
>>> import numpy as np
```

```
>>> class NDDataMask1D(NDDataRef):
...     def _slice_mask(self, item):
...         # Multidimensional slices are represented by tuples:
...         if isinstance(item, tuple):
...             # only use the first dimension of the slice
...             return self.mask[item[0]]
...         # Let the superclass deal with the other cases
...         return super()._slice_mask(item)
```

```
>>> ndd = NDDataMask1D(np.ones((3,3)), mask=np.ones(3, dtype=bool))
>>> nddsliced = ndd[1:3,1:3]
>>> nddsliced.mask
array([ True,  True]...)
```

Note

The methods slicing the attributes are prefixed by a `_slice_*` where `*` can be `mask`, `uncertainty`, or `wcs`. So overriding them is the most convenient way to customize how the attributes are sliced.

Note

If slicing should affect the `unit` or `meta` see the next example.

Slicing an Additional Property

Building on the added property `flags`, we want them to be sliceable:

```
>>> class NDDataWithFlags(NDDataRef):
...     def __init__(self, *args, **kwargs):
...         # Remove flags attribute if given and pass it to the
setter.
...         self.flags = kwargs.pop('flags') if 'flags' in kwargs
else None
...         super().__init__(*args, **kwargs)
...
...     @property
...     def flags(self):
...         return self._flags
...
...     @flags.setter
...     def flags(self, value):
...         self._flags = value
...
...     def _slice(self, item):
...         # slice all normal attributes
...         kwargs = super()._slice(item)
...         # The arguments for creating a new instance are saved in
kwargs
...         # so we need to add another keyword "flags" and add the
sliced flags
...         kwargs['flags'] = self.flags[item]
...         return kwargs # these must be returned
```

```
>>> ndd = NDDataWithFlags([1,2,3], flags=[0, 0.2, 0.3])
>>> ndd2 = ndd[1:3]
>>> ndd2.flags
[0.2, 0.3]
```

If you wanted to keep just the original `flags` instead of the sliced ones, you could use `kwargs['flags'] = self.flags` and omit the `[item]`.

NDDataBase

The class **NDDataBase** is a metaclass — when subclassing it, all properties of **NDDataBase** *must* be overridden in the subclass.

Subclassing from **NDDataBase** gives you complete flexibility in how you implement data storage and the other properties. If your data is stored in a `numpy` array (or something that behaves like a `numpy` array), it may be more convenient to subclass **NDData** instead of **NDDataBase**.

Example
To implement the NDDataBase interface by creating a read-only container:

```
>>> from astropy.nddata import NDDataBase

>>> class NDDataReadOnlyNoRestrictions(NDDataBase):
...     def __init__(self, data, unit, mask, uncertainty, meta, wcs):
...         self._data = data
...         self._unit = unit
...         self._mask = mask
...         self._uncertainty = uncertainty
...         self._meta = meta
...         self._wcs = wcs
...
...     @property
...     def data(self):
...         return self._data
...
...     @property
...     def unit(self):
...         return self._unit
...
...     @property
...     def mask(self):
...         return self._mask
...
...     @property
...     def uncertainty(self):
...         return self._uncertainty
...
...     @property
...     def meta(self):
...         return self._meta
...
...     @property
...     def wcs(self):
...         return self._wcs
```

```
>>> # A meaningless test to show that creating this class is
possible:
>>> NDDataReadOnlyNoRestrictions(1,2,3,4,5,6) is not None
True
```

Note

Actually defining an `__init__` is not necessary and the properties could return arbitrary values but the properties **must** be defined.

Subclassing `NDUncertainty`

Warning

The internal interface of NDUncertainty and subclasses is experimental and might change in future versions.

Subclasses deriving from `NDUncertainty` need in order to implement:

- Property `uncertainty_type` should return a string describing the uncertainty, for example, `"ivar"` for inverse variance.
- Methods for propagation: **`_propagate_*`** where `*` is the name of the universal function (ufunc) that is used on the `NDData` parent.

Creating an Uncertainty without Propagation

`UnknownUncertainty` is a minimal working implementation without error propagation. We can create an uncertainty by storing systematic uncertainties:

```
>>> from astropy.nddata import NDUncertainty

>>> class SystematicUncertainty(NDUncertainty):
...     @property
...     def uncertainty_type(self):
...         return 'systematic'
...
...     def _data_unit_to_uncertainty_unit(self, value):
...         return None
...
...     def _propagate_add(self, other_uncert, *args, **kwargs):
...         return None
...
...     def _propagate_subtract(self, other_uncert, *args, **kwargs):
...         return None
...
...     def _propagate_multiply(self, other_uncert, *args, **kwargs):
```

```
...             return None
...
...         def _propagate_divide(self, other_uncert, *args, **kwargs):
...             return None

>>> SystematicUncertainty([10])
SystematicUncertainty([10])
```

Performance Tips

- Using the uncertainty class `VarianceUncertainty` will be somewhat more efficient than the other two uncertainty classes, `InverseVariance` and `StdDevUncertainty`. The latter two are converted to variance for the purposes of error propagation and then converted from variance back to the original uncertainty type. The performance difference should be small.
- When possible, mask values by setting them to `np.nan` and use the `numpy` functions and methods that automatically exclude `np.nan`, like `np.nanmedian` and `np.nanstd`. This will typically be much faster than using `numpy.ma.MaskedArray`.

Reference/API

astropy.nddata Package

The `astropy.nddata` subpackage provides the `NDData` class and related tools to manage n-dimensional array-based data (e.g. CCD images, IFU Data, grid-based simulation data, …). This is more than just `numpy.ndarray` objects, because it provides metadata that cannot be easily provided by a single array.

Functions

`add_array`(array_large, array_small, position)	Add a smaller array at a given position in a larger array.
`bitfield_to_boolean_mask`(bitfield[, …])	Converts an array of bit fields to a boolean (or integer) mask array according to a bit mask constructed from the supplied bit flags (see `ignore_flags` parameter).
`block_reduce`(data, block_size[, func])	Downsample a data array by applying a function to local blocks.
`block_replicate`(data, block_size[, conserve_sum])	Upsample a data array by block replication.
`extend_bit_flag_map`(cls_name[, base_cls])	A convenience function for creating bit flags maps by subclassing an existing map and adding additional flags supplied as keyword

	arguments.
extract_array(array_large, shape, position)	Extract a smaller array of the given shape and position from a larger array.
fits_ccddata_reader(filename[, hdu, unit, …])	Generate a CCDData object from a FITS file.
fits_ccddata_writer(ccd_data, filename[, …])	Write CCDData object to FITS file.
interpret_bit_flags(bit_flags[, flip_bits, …])	Converts input bit flags to a single integer value (bit mask) or **None**.
overlap_slices(large_array_shape, …[, mode])	Get slices for the overlapping part of a small and a large array.
reshape_as_blocks(data, block_size)	Reshape a data array into blocks.
subpixel_indices(position, subsampling)	Convert decimal points to indices, given a subsampling factor.
support_nddata([_func, accepts, repack, …])	Decorator to wrap functions that could accept an NDData instance with its properties passed as function arguments.

Classes

BitFlagNameMap()	A base class for bit flag name maps used to describe data quality (DQ) flags of images by provinding a mapping from a mnemonic flag name to a flag value.
CCDData(*args, **kwd)	A class describing basic CCD data.
Conf()	Configuration parameters for **astropy.nddata**.
Cutout2D(data, position, size[, wcs, mode, …])	Create a cutout object from a 2D array.
FlagCollection(*args, **kwargs)	The purpose of this class is to provide a dictionary for containing arrays of flags for the **NDData** class.
IncompatibleUncertaintiesException	This exception should be used to indicate cases in which uncertainties with two different classes can not be propagated.
InvalidBitFlag	Indicates that a value is not an integer that is a power of 2.
InverseVariance([array, copy, unit])	Inverse variance uncertainty assuming first order Gaussian error propagation.
MissingDataAssociationException	This exception should be used to indicate that an uncertainty instance has not been associated with a parent **NDData** object.
NDArithmeticMixin()	Mixin class to add arithmetic to an NDData object.
NDData(data[, uncertainty, mask, wcs, meta, …])	A container for **numpy.ndarray**-based datasets, using the **NDDataBase** interface.
NDDataArray(data, *args[, flags])	An `NDData` object with arithmetic.

NDDataBase()	Base metaclass that defines the interface for N-dimensional datasets with associated meta information used in `astropy`.
NDDataRef(data[, uncertainty, mask, wcs, …])	Implements **NDData** with all Mixins.
NDIOMixin()	Mixin class to connect NDData to the astropy input/output registry.
NDSlicingMixin()	Mixin to provide slicing on objects using the **NDData** interface.
NDUncertainty([array, copy, unit])	This is the metaclass for uncertainty classes used with **NDData**.
NoOverlapError	Raised when determining the overlap of non-overlapping arrays.
PartialOverlapError	Raised when arrays only partially overlap.
StdDevUncertainty([array, copy, unit])	Standard deviation uncertainty assuming first order gaussian error propagation.
UnknownUncertainty([array, copy, unit])	This class implements any unknown uncertainty type.
VarianceUncertainty([array, copy, unit])	Variance uncertainty assuming first order Gaussian error propagation.

astropy.nddata.bitmask Module

A module that provides functions for manipulating bit masks and data quality (DQ) arrays.

Functions

bitfield_to_boolean_mask(bitfield[, …])	Converts an array of bit fields to a boolean (or integer) mask array according to a bit mask constructed from the supplied bit flags (see `ignore_flags` parameter).
interpret_bit_flags(bit_flags[, flip_bits, …])	Converts input bit flags to a single integer value (bit mask) or **None**.
extend_bit_flag_map(cls_name[, base_cls])	A convenience function for creating bit flags maps by subclassing an existing map and adding additional flags supplied as keyword arguments.

Classes

BitFlagNameMap()	A base class for bit flag name maps used to describe data quality (DQ) flags of images by provinding a mapping from a mnemonic flag name to a flag value.
InvalidBitFlag	Indicates that a value is not an integer that is a power of 2.

astropy.nddata.utils Module

This module includes helper functions for array operations.

Functions

`extract_array`(array_large, shape, position)	Extract a smaller array of the given shape and position from a larger array.
`add_array`(array_large, array_small, position)	Add a smaller array at a given position in a larger array.
`subpixel_indices`(position, subsampling)	Convert decimal points to indices, given a subsampling factor.
`overlap_slices`(large_array_shape, …[, mode])	Get slices for the overlapping part of a small and a large array.

Classes

`NoOverlapError`	Raised when determining the overlap of non-overlapping arrays.
`PartialOverlapError`	Raised when arrays only partially overlap.
`Cutout2D`(data, position, size[, wcs, mode, …])	Create a cutout object from a 2D array.

Data Tables (`astropy.table`)

Introduction

`astropy.table` provides functionality for storing and manipulating heterogeneous tables of data in a way that is familiar to `numpy` users. A few notable capabilities of this package are:

- Initialize a table from a wide variety of input data structures and types.
- Modify a table by adding or removing columns, changing column names, or adding new rows of data.
- Handle tables containing missing values.
- Include table and column metadata as flexible data structures.
- Specify a description, units, and output formatting for columns.
- Interactively scroll through long tables similar to using `more`.
- Create a new table by selecting rows or columns from a table.
- Perform Table Operations like database joins, concatenation, and binning.
- Maintain a table index for fast retrieval of table items or ranges.
- Manipulate multidimensional columns.
- Handle non-native (mixin) column types within table.
- Methods for Reading and Writing Table Objects to files.
- Hooks for Subclassing Table and its component classes.

Getting Started

The basic workflow for creating a table, accessing table elements, and modifying the table is shown below. These examples demonstrate a concise case, while the full `astropy.table` documentation is available from the Using table section.

First create a simple table with columns of data named `a`, `b`, `c`, and `d`. These columns have integer, float, string, and `Quantity` values respectively:

```
>>> from astropy.table import QTable
>>> import astropy.units as u
>>> import numpy as np

>>> a = np.array([1, 4, 5], dtype=np.int32)
>>> b = [2.0, 5.0, 8.5]
>>> c = ['x', 'y', 'z']
>>> d = [10, 20, 30] * u.m / u.s

>>> t = QTable([a, b, c, d],
...            names=('a', 'b', 'c', 'd'),
...            meta={'name': 'first table'})
```

Comments:

- Column `a` is a `numpy` array with a specified `dtype` of `int32`. If the data type is not provided, the default type for integers is `int64` on Mac and Linux and `int32` on Windows.
- Column `b` is a list of `float` values, represented as `float64`.
- Column `c` is a list of `str` values, represented as unicode. See Bytestring Columns for more information.
- Column `d` is a `Quantity` array. Since we used `QTable`, this stores a native `Quantity` within the table and brings the full power of Units and Quantities (astropy.units) to this column in the table.

Note

If the table data have no units or you prefer to not use `Quantity`, then you can use the `Table` class to create tables. The **only** difference between `QTable` and `Table` is the behavior when adding a column that has units. See Quantity and QTable and Columns with Units for details on the differences and use cases.

There are many other ways of Constructing a Table, including from a list of rows (either tuples or dicts), from a `numpy` structured or 2D array, by adding columns or rows incrementally, or even from a `pandas.DataFrame`.

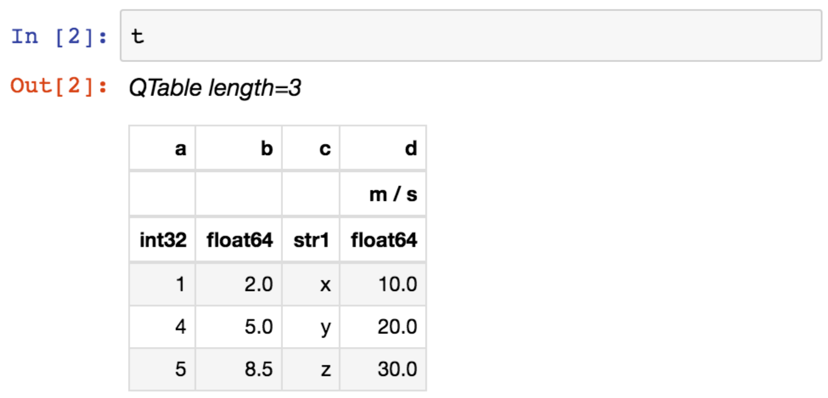
There are a few ways of Accessing a Table. You can get detailed information about the table values and column definitions as follows:

```
>>> t
<QTable length=3>
  a      b     c      d
                    m / s
int32 float64 str1 float64
----- ------- ---- -------
    1     2.0    x    10.0
    4     5.0    y    20.0
    5     8.5    z    30.0
```

You can get summary information about the table as follows:

```
>>> t.info
<QTable length=3>
name  dtype   unit  class
---- ------- ----- --------
   a   int32         Column
   b float64         Column
   c    str1         Column
   d float64 m / s Quantity
```

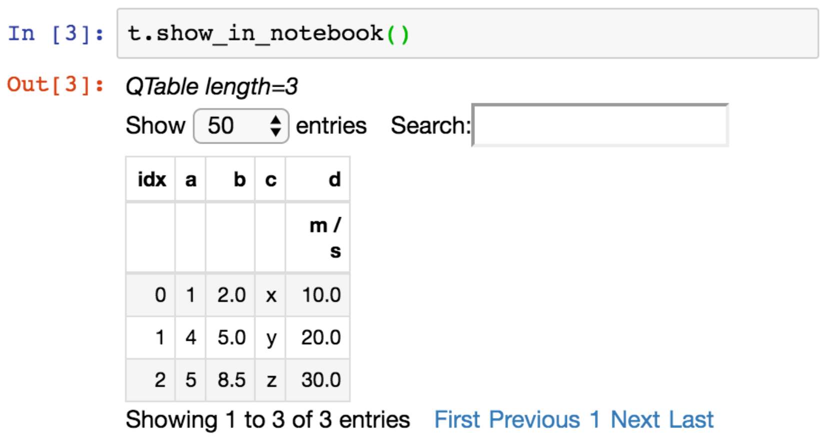
From within a Jupyter notebook, the table is displayed as a formatted HTML table (details of how it appears can be changed by altering the `astropy.table.default_notebook_table_class` configuration item):



```
In [2]: t
Out[2]: QTable length=3
```

a	b	c	d
			m / s
int32	float64	str1	float64
1	2.0	x	10.0
4	5.0	y	20.0
5	8.5	z	30.0

Or you can get a fancier notebook interface with in-browser search, and sort using **`show_in_notebook`**:



```
In [3]: t.show_in_notebook()
```

Out[3]: *QTable length=3*

Show 50 entries Search:

idx	a	b	c	d
				m / s
0	1	2.0	x	10.0
1	4	5.0	y	20.0
2	5	8.5	z	30.0

Showing 1 to 3 of 3 entries First Previous 1 Next Last

If you print the table (either from the notebook or in a text console session) then a formatted version appears:

```
>>> print(t)
 a   b   c    d
            m / s
--- --- --- -----
  1 2.0   x  10.0
  4 5.0   y  20.0
  5 8.5   z  30.0
```

If you do not like the format of a particular column, you can change it:

```
>>> t['b'].info.format = '7.3f'
>>> print(t)
 a     b     c    d
                m / s
--- ------- --- -----
  1   2.000   x  10.0
  4   5.000   y  20.0
  5   8.500   z  30.0
```

For a long table you can scroll up and down through the table one page at time:

```
>>> t.more()
```

You can also display it as an HTML-formatted table in the browser:

```
>>> t.show_in_browser()
```

Or as an interactive (searchable and sortable) javascript table:

```
>>> t.show_in_browser(jsviewer=True)
```

Now examine some high-level information about the table:

```
>>> t.colnames
['a', 'b', 'c', 'd']
>>> len(t)
3
>>> t.meta
{'name': 'first table'}
```

Access the data by column or row using familiar `numpy` structured array syntax:

```
>>> t['a']       # Column 'a'
<Column name='a' dtype='int32' length=3>
1
4
5

>>> t['a'][1]    # Row 1 of column 'a'
4

>>> t[1]         # Row object for table row index=1
<Row index=1>
  a      b     c      d
                    m / s
int32 float64 str1 float64
----- ------- ---- -------
    4   5.000    y    20.0


>>> t[1]['a']    # Column 'a' of row 1
4
```

You can retrieve a subset of a table by rows (using a slice) or by columns (using column names), where the subset is returned as a new table:

```
>>> print(t[0:2])      # Table object with rows 0 and 1
 a     b     c    d
                m / s
--- ------- --- -----
  1   2.000   x  10.0
  4   5.000   y  20.0


>>> print(t['a', 'c'])  # Table with cols 'a', 'c'
 a   c
--- ---
```

```
  1   x
  4   y
  5   z
```

Modifying a Table in place is flexible and works as you would expect:

```
>>> t['a'][:] = [-1, -2, -3]    # Set all column values in place
>>> t['a'][2] = 30              # Set row 2 of column 'a'
>>> t[1] = (8, 9.0, "W", 4 * u.m / u.s) # Set all row values
>>> t[1]['b'] = -9              # Set column 'b' of row 1
>>> t[0:2]['b'] = 100.0         # Set column 'b' of rows 0 and 1
>>> print(t)
 a     b     c    d
                m / s
--- ------- --- -----
 -1 100.000   x  10.0
  8 100.000   W   4.0
 30   8.500   z  30.0
```

Replace, add, remove, and rename columns with the following:

```
>>> t['b'] = ['a', 'new', 'dtype']   # Replace column b (different
from in-place)
>>> t['e'] = [1, 2, 3]               # Add column d
>>> del t['c']                       # Delete column c
>>> t.rename_column('a', 'A')        # Rename column a to A
>>> t.colnames
['A', 'b', 'd', 'e']
```

Adding a new row of data to the table is as follows. Note that the unit value is given in `cm / s` but will be added to the table as `0.1 m / s` in accord with the existing unit.

```
>>> t.add_row([-8, 'string', 10 * u.cm / u.s, 10])
>>> len(t)
4
```

You can create a table with support for missing values, for example, by setting `masked=True`:

```
>>> t = QTable([a, b, c], names=('a', 'b', 'c'), masked=True, dtype=
('i4', 'f8', 'U1'))
>>> t['a'].mask = [True, True, False]
>>> t
<QTable masked=True length=3>
  a      b     c
```

```
int32 float64 str1
----- ------- ----
   --     2.0    x
   --     5.0    y
    5     8.5    z
```

In addition to **Quantity**, you can include certain object types like **Time**, **SkyCoord**, and **NdarrayMixin** in your table. These “mixin” columns behave like a hybrid of a regular **Column** and the native object type (see Mixin Columns). For example:

```
>>> from astropy.time import Time
>>> from astropy.coordinates import SkyCoord
>>> tm = Time(['2000:002', '2002:345'])
>>> sc = SkyCoord([10, 20], [-45, +40], unit='deg')
>>> t = QTable([tm, sc], names=['time', 'skycoord'])
>>> t
<QTable length=2>
         time          skycoord
                       deg,deg
        object          object
--------------------- ----------
2000:002:00:00:00.000 10.0,-45.0
2002:345:00:00:00.000  20.0,40.0
```

Now let us compute the interval since the launch of the Chandra X-ray Observatory aboard STS-93 and store this in our table as a **Quantity** in days:

```
>>> dt = t['time'] - Time('1999-07-23 04:30:59.984')
>>> t['dt_cxo'] = dt.to(u.d)
>>> t['dt_cxo'].info.format = '.3f'
>>> print(t)
         time          skycoord   dt_cxo
                       deg,deg      d
--------------------- ---------- --------
2000:002:00:00:00.000 10.0,-45.0  162.812
2002:345:00:00:00.000  20.0,40.0 1236.812
```

Using `table`

The details of using **astropy.table** are provided in the following sections:

Construct Table

Constructing a Table

There is great deal of flexibility in the way that a table can be initially constructed. Details on the inputs to the `Table` and `QTable` constructors are in the Initialization Details section. However, the best way to understand how to make a table is by example.

Examples

Much of the flexibility lies in the types of data structures which can be used to initialize the table data. The examples below show how to create a table from scratch with no initial data, as well as create a table with a list of columns, a dictionary of columns, or from `numpy` arrays (either structured or homogeneous).

Setup

For the following examples you need to import the `QTable`, `Table`, and `Column` classes along with the Units and Quantities (astropy.units) package and the `numpy` package:

```
>>> from astropy.table import QTable, Table, Column
>>> from astropy import units as u
>>> import numpy as np
```

Creating from Scratch

A Table can be created without any initial input data or even without any initial columns. This is useful for building tables dynamically if the initial size, columns, or data are not known.

> **Note**
>
> Adding rows requires making a new copy of the entire table each time, so in the case of large tables this may be slow. On the other hand, adding columns is fast.

```
>>> t = Table()
>>> t['a'] = [1, 4]
>>> t['b'] = [2.0, 5.0]
>>> t['c'] = ['x', 'y']

>>> t = Table(names=('a', 'b', 'c'), dtype=('f4', 'i4', 'S2'))
>>> t.add_row((1, 2.0, 'x'))
>>> t.add_row((4, 5.0, 'y'))

>>> t = Table(dtype=[('a', 'f4'), ('b', 'i4'), ('c', 'S2')])
```

If your data columns have physical units associated with them then we recommend using the **QTable** class. This will allow the column to be stored in the table as a native **Quantity** and bring the full power of Units and Quantities (astropy.units) to the table.

```
>>> t = QTable()
>>> t['a'] = [1, 4]
>>> t['b'] = [2.0, 5.0] * u.cm / u.s
>>> t['c'] = ['x', 'y']
>>> type(t['b'])
<class 'astropy.units.quantity.Quantity'>
```

List of Columns

A typical case is where you have a number of data columns with the same length defined in different variables. These might be Python lists or `numpy` arrays or a mix of the two. These can be used to create a **Table** by putting the column data variables into a Python list. In this case the column names are not defined by the input data, so they must either be set using the `names` keyword or they will be automatically generated as `col<N>`.

```
>>> a = np.array([1, 4], dtype=np.int32)
>>> b = [2.0, 5.0]
>>> c = ['x', 'y']
>>> t = Table([a, b, c], names=('a', 'b', 'c'))
>>> t
<Table length=2>
  a      b     c
int32 float64 str1
----- ------- ----
    1     2.0    x
    4     5.0    y
```

Make a new table using columns from the first table

Once you have a **Table**, then you can make a new table by selecting columns and putting this into a Python list (e.g., `[ t['c'], t['a'] ]`):

```
>>> Table([t['c'], t['a']])
<Table length=2>
 c     a
str1 int32
---- -----
   x     1
   y     4
```

Make a new table using expressions involving columns

The **Column** object is derived from the standard `numpy` array and can be used directly in arithmetic expressions. This allows for a compact way of making a new table with modified column values:

```
>>> Table([t['a']**2, t['b'] + 10])
<Table length=2>
  a      b
int32 float64
----- -------
    1    12.0
   16    15.0
```

Different types of column data

The list input method for **Table** is very flexible since you can use a mix of different data types to initialize a table:

```
>>> a = (1, 4)
>>> b = np.array([[2, 3], [5, 6]])  # vector column
>>> c = Column(['x', 'y'], name='axis')
>>> arr = (a, b, c)
>>> Table(arr)
<Table length=2>
 col0 col1 [2] axis
int64  int64   str1
----- -------- ----
    1   2 .. 3    x
    4   5 .. 6    y
```

Notice that in the third column the existing column name `'axis'` is used.

Dict of Columns

A dictionary of column data can be used to initialize a **Table**.

```
>>> arr = {'a': np.array([1, 4], dtype=np.int32),
...        'b': [2.0, 5.0],
...        'c': ['x', 'y']}
>>>
>>> Table(arr)
<Table length=2>
  a    c      b
int32 str1 float64
----- ---- -------
    1    x     2.0
    4    y     5.0
```

Specify the column order and optionally the data types

```
>>> Table(arr, names=('a', 'b', 'c'), dtype=('f8', 'i4', 'S2'))
<Table length=2>
   a      b     c
float64 int32 str2
------- ----- ----
    1.0     2    x
    4.0     5    y
```

Different types of column data

The input column data can be any data type that can initialize a **Column** object:

```
>>> arr = {'a': (1, 4),
...        'b': np.array([[2, 3], [5, 6]]),
...        'c': Column(['x', 'y'], name='axis')}
>>> Table(arr, names=('a', 'b', 'c'))
<Table length=2>
  a    b [2]    c
int64 int64  str1
----- ------ ----
    1 2 .. 3    x
    4 5 .. 6    y
```

Notice that the key `'c'` takes precedence over the existing column name `'axis'` in the third column. Also see that the `'b'` column is a vector column where each row element is itself a two-element array.

Renaming columns is not possible

```
>>> Table(arr, names=('a_new', 'b_new', 'c_new'))
Traceback (most recent call last):
  ...
KeyError: 'a_new'
```

Row Data

Row-oriented data can be used to create a table using the `rows` keyword argument.

List of data records as list or tuple

If you have row-oriented input data such as a list of records, you need to use the `rows` keyword to create a table:

```
>>> data_rows = [(1, 2.0, 'x'),
...              (4, 5.0, 'y'),
...              (5, 8.2, 'z')]
>>> t = Table(rows=data_rows, names=('a', 'b', 'c'))
>>> print(t)
```

```
 a   b   c
--- --- ---
  1 2.0   x
  4 5.0   y
  5 8.2   z
```

The data object passed as the `rows` argument can be any form which is parsable by the `np.rec.fromrecords()` function.

List of dict objects

You can also initialize a table with row values. This is constructed as a list of dict objects. The keys determine the column names:

```
>>> data = [{'a': 5, 'b': 10},
...         {'a': 15, 'b': 20}]
>>> t = Table(rows=data)
>>> print(t)
 a   b
--- ---
  5  10
 15  20
```

If there are missing keys in one or more rows then the corresponding values will be marked as missing (masked):

```
>>> t = Table(rows=[{'a': 5, 'b': 10}, {'a': 15, 'c': 50}])
>>> print(t)
 a   b   c
--- --- ---
  5  10  --
 15  --  50
```

You can also preserve the column order by using `OrderedDict`. If the first item is an `OrderedDict` then the order is preserved:

```
>>> from collections import OrderedDict
>>> row1 = OrderedDict([('b', 1), ('a', 0)])
>>> row2 = OrderedDict([('b', 11), ('a', 10)])
>>> rows = [row1, row2]
>>> Table(rows=rows, dtype=('i4', 'i4'))
<Table length=2>
  b     a
int32 int32
----- -----
    1     0
   11    10
```

Single row

You can also make a new table from a single row of an existing table:

```
>>> a = [1, 4]
>>> b = [2.0, 5.0]
>>> t = Table([a, b], names=('a', 'b'))
>>> t2 = Table(rows=t[1])
```

Remember that a **Row** has effectively a zero length compared to the newly created **Table** which has a length of one. This is similar to the difference between a scalar `1` (length 0) and an array such as `np.array([1])` with length 1.

Note

In the case of input data as a list of dicts or a single **Table** row, you can supply the data as the `data` argument since these forms are always unambiguous. For example, `Table([{'a': 1}, {'a': 2}])` is accepted. However, a list of records must always be provided using the `rows` keyword, otherwise it will be interpreted as a list of columns.

NumPy Structured Array

The structured array is the standard mechanism in `` `numpy` `` for storing heterogeneous table data. Most scientific I/O packages that read table files (e.g., **astropy.io.fits**, **astropy.io.votable**, and asciitable) will return the table in an object that is based on the structured array. A structured array can be created using:

```
>>> arr = np.array([(1, 2.0, 'x'),
...                 (4, 5.0, 'y')],
...                dtype=[('a', 'i4'), ('b', 'f8'), ('c', 'S2')])
```

From `arr` it is possible to create the corresponding **Table** object:

```
>>> Table(arr)
<Table length=2>
  a      b     c
int32 float64 str2
----- ------- ----
    1     2.0    x
    4     5.0    y
```

Note that in the above example and most the following examples we are creating a table and immediately asking the interactive Python interpreter to print the table to see what we made. In real code you might do something like:

```
>>> table = Table(arr)
>>> print(table)
 a   b   c
--- --- ---
  1 2.0   x
  4 5.0   y
```

New column names

The column names can be changed from the original values by providing the `names` argument:

```
>>> Table(arr, names=('a_new', 'b_new', 'c_new'))
<Table length=2>
a_new  b_new  c_new
int32 float64  str2
----- ------- -----
    1     2.0     x
    4     5.0     y
```

New data types

The data type for each column can likewise be changed with `dtype`:

```
>>> Table(arr, dtype=('f4', 'i4', 'S4'))
<Table length=2>
   a       b     c
float32 int32 str4
------- ----- ----
    1.0     2    x
    4.0     5    y

>>> Table(arr, names=('a_new', 'b_new', 'c_new'), dtype=('f4', 'i4',
'S4'))
<Table length=2>
 a_new  b_new c_new
float32 int32  str4
------- ----- -----
    1.0     2     x
    4.0     5     y
```

NumPy Homogeneous Array

A `` `numpy` `` 1D array is treated as a single row table where each element of the array corresponds to a column:

```
>>> Table(np.array([1, 2, 3]), names=['a', 'b', 'c'], dtype=('i8',
'i8', 'i8'))
```

```
<Table length=1>
  a     b     c
int64 int64 int64
----- ----- -----
    1     2     3
```

A `` `numpy` `` 2D array (where all elements have the same type) can also be converted into a **`Table`**. In this case the column names are not specified by the data and must either be provided by the user or will be automatically generated as `col<N>` where `<N>` is the column number.

Basic example with automatic column names

```
>>> arr = np.array([[1, 2, 3],
...                 [4, 5, 6]], dtype=np.int32)
>>> Table(arr)
<Table length=2>
 col0  col1  col2
int32 int32 int32
----- ----- -----
    1     2     3
    4     5     6
```

Column names and types specified

```
>>> Table(arr, names=('a_new', 'b_new', 'c_new'), dtype=('f4', 'i4', 
'S4'))
<Table length=2>
 a_new  b_new c_new
float32 int32  str4
------- ----- -----
    1.0     2     3
    4.0     5     6
```

Referencing the original data

It is possible to reference the original data for a homogeneous array as long as the data types are not changed:

```
>>> t = Table(arr, copy=False)
```

Python arrays versus ```numpy``` arrays as input

There is a slightly subtle issue that is important to understand about the way that **`Table`** objects are created. Any data input that looks like a Python list (including a tuple) is considered to be a list of columns. In contrast, a homogeneous `` `numpy` `` array input is interpreted as a list of rows:

```
>>> arr = [[1, 2, 3],
...        [4, 5, 6]]
>>> np_arr = np.array(arr)

>>> print(Table(arr))    # Two columns, three rows
col0 col1
---- ----
   1    4
   2    5
   3    6

>>> print(Table(np_arr))  # Three columns, two rows
col0 col1 col2
---- ---- ----
   1    2    3
   4    5    6
```

This dichotomy is needed to support flexible list input while retaining the natural interpretation of 2D `` `numpy` `` arrays where the first index corresponds to data "rows" and the second index corresponds to data "columns."

From an Existing Table

A new table can be created by selecting a subset of columns in an existing table:

```
>>> t = Table(names=('a', 'b', 'c'))
>>> t['c', 'b', 'a']  # Makes a copy of the data
<Table length=0>
   c       b       a
float64 float64 float64
------- ------- -------
```

An alternate way to use the `columns` attribute (explained in the TableColumns section) to initialize a new table. This lets you choose columns by their numerical index or name and supports slicing syntax:

```
>>> Table(t.columns[0:2])
<Table length=0>
   a       b
float64 float64
------- -------

>>> Table([t.columns[0], t.columns['c']])
<Table length=0>
   a       c
float64 float64
------- -------
```

To create a copy of an existing table that is empty (has no rows):

```
>>> t = Table([[1.0, 2.3], [2.1, 3]], names=['x', 'y'])
>>> t
<Table length=2>
   x       y
float64 float64
------- -------
    1.0     2.1
    2.3     3.0

>>> tcopy = t[:0].copy()
>>> tcopy
<Table length=0>
   x       y
float64 float64
------- -------
```

Empty Array of a Known Size

If you do know the size that your table will be, but do not know the values in advance, you can create a zeroed `numpy` array and build the `astropy` table from it:

```
>>> N = 3
>>> dtype = [('a', 'i4'), ('b', 'f8'), ('c', 'bool')]
>>> t = Table(data=np.zeros(N, dtype=dtype))
>>> t
<Table length=3>
  a      b      c
int32 float64  bool
----- ------- -----
    0     0.0 False
    0     0.0 False
    0     0.0 False
```

For example, you can then fill in this table row by row with values extracted from another table, or generated on the fly:

```
>>> for i in range(len(t)):
...     t[i] = (i, 2.5*i, i % 2)
>>> t
<Table length=3>
  a      b      c
int32 float64  bool
----- ------- -----
    0     0.0 False
    1     2.5  True
    2     5.0 False
```

Pandas DataFrame

The section on Interfacing with the Pandas Package gives details on how to initialize a **`Table`** using a **`pandas.DataFrame`** via the **`from_pandas`** class method. This provides a convenient way to take advantage of the many I/O and table manipulation methods in pandas.

Comment Lines

Comment lines in an ASCII file can be added via the `'comments'` key in the table's metadata. The following will insert two comment lines in the output ASCII file unless `comment=False` is explicitly set in `write()`:

```
>>> import sys
>>> from astropy.table import Table
>>> t = Table(names=('a', 'b', 'c'), dtype=('f4', 'i4', 'S2'))
>>> t.add_row((1, 2.0, 'x'))
>>> t.meta['comments'] = ['Here is my explanatory text. This is
awesome.',
...                           'Second comment line.']
>>> t.write(sys.stdout, format='ascii')
# Here is my explanatory text. This is awesome.
# Second comment line.
a b c
1.0 2 x
```

Initialization Details

A table object is created by initializing a **`Table`** class object with the following arguments, all of which are optional:

`data` : *numpy ndarray, dict, list, Table, or table-like object, optional*

Data to initialize table.

`masked` : *bool, optional*

Specify whether the table is masked.

`names` : *list, optional*
Specify column names.

`dtype` : *list, optional*
Specify column data types.

`meta` : *dict, optional*
Metadata associated with the table.

`copy` : *bool, optional*
Copy the input data. If the input is a Table the `meta` is always copied regardless of the `copy` parameter. Default is True.

`rows` : *numpy ndarray, list of lists, optional*
Row-oriented data for table instead of `data` argument.

`copy_indices` : *bool, optional*
Copy any indices in the input data. Default is True.

`units` : *list, dict, optional*
List or dict of units to apply to columns.

`descriptions` : *list, dict, optional*
List or dict of descriptions to apply to columns.

`**kwargs` : *dict, optional*
Additional keyword args when converting table-like object.

The following subsections provide further detail on the values and options for each of the keyword arguments that can be used to create a new **`Table`** object.

data

The **`Table`** object can be initialized with several different forms for the `data` argument.

``numpy`` ndarray (structured array)

The base column names are the field names of the `data` structured array. The `names` list (optional) can be used to select particular fields and/or reorder the base names. The `dtype` list (optional) must match the length of `names` and is used to override the existing `data` types.

``numpy`` ndarray (homogeneous)

If the `data` ndarray is one-dimensional then it is treated as a single row table where each element of the array corresponds to a column.

If the `data` ndarray is at least two-dimensional, then the first (left-most) index corresponds to row number (table length) and the second index corresponds to column number (table width). Higher dimensions get absorbed in the shape of each table cell.

If provided, the `names` list must match the “width” of the `data` argument. The default for `names` is to auto-generate column names in the form `col<N>`. If provided, the `dtype` list overrides the base column types and must match the length of `names`.

dict-like

The keys of the `data` object define the base column names. The corresponding values can be `Column` objects, `numpy` arrays, or list- like objects. The `names` list (optional) can be used to select particular fields and/or reorder the base names. The `dtype` list (optional) must match the length of `names` and is used to override the existing or default data types.

list-like

Each item in the `data` list provides a column of data values and can be a `Column` object, `numpy` array, or list-like object. The `names` list defines the name of each column. The names will be auto-generated if not provided (either from the `names` argument or by `Column` objects). If provided, the `names` argument must match the number of items in the `data` list. The optional `dtype` list will override the existing or default data types and must match `names` in length.

list-of-dicts

Similar to Python’s built-in `csv.DictReader`, each item in the `data` list provides a row of data values and must be a dict. The key values in each dict define the column names and each row must have identical column names. The `names` argument may be supplied to specify column ordering. If it is not provided, the column order will default to alphabetical. If the first item is an `OrderedDict`, then the column order is preserved. The `dtype` list may be specified, and must correspond to the order of output columns. If any row’s keys do not match the rest of the rows, a ValueError will be thrown.

table-like object

If another table-like object has a `__astropy_table__` method then that object can be used to directly create a `Table` object. See the Table-like objects section for details.

None

Initialize a zero-length table. If `names` and optionally `dtype` are provided, then the corresponding columns are created.

names

The `names` argument provides a way to specify the table column names or override the existing ones. By default, the column names are either taken from existing names (for `ndarray` or `Table` input) or auto-generated as `col<N>`. If `names` is provided, then it must be a list with the same length as the number of columns. Any list elements with value `None` fall back to the default name.

In the case where `data` is provided as a dict of columns, the `names` argument can be supplied to specify the order of columns. The `names` list must then contain each of the keys in the `data` dict. If `names` is not supplied, then the order of columns in the output table is not determinate.

dtype

The `dtype` argument provides a way to specify the table column data types or override the existing types. By default, the types are either taken from existing types (for `ndarray` or `Table` input) or auto-generated by the `numpy.array()` routine. If `dtype` is provided then it must be a list with the same length as the number of columns. The values must be valid `numpy.dtype` initializers or `None`. Any list elements with value `None` fall back to the default type.

In the case where `data` is provided as a dict of columns, the `dtype` argument must be accompanied by a corresponding `names` argument in order to uniquely specify the column ordering.

meta

The `meta` argument is an object that contains metadata associated with the table. It is recommended that this object be a dict or OrderedDict, but the only firm requirement is that it can be copied with the standard library `copy.deepcopy()` routine. By default, `meta` is an empty OrderedDict.

copy

By default, the input `data` are copied into a new internal `np.ndarray` object in the `Table` object. In the case where `data` is either an `np.ndarray` object, a `dict`, or an existing `Table`, it is possible to use a reference to the existing data by setting `copy=False`. This has the advantage of reducing memory use and being faster. However, you should take care because any modifications to the new `Table` data will also be seen in the original input data. See the Copy versus Reference section for more information.

rows

This argument allows for providing data as a sequence of rows, in contrast to the `data` keyword, which generally assumes data are a sequence of columns. The Row data section provides details.

copy_indices
If you are initializing a table from another table that has table indices defined, then this option allows copying that table *without* copying the indices by setting `copy_indices=False`. By default, the indices are copied.

units
This allows for setting the unit for one or more columns at the time of creating the table. The input can be either a list of unit values corresponding to each of the columns in the table (using `None` or `''` for no unit), or a `dict` that provides the unit for specified column names. For example:

```
>>> from astropy.table import QTable
>>> dat = [[1, 2], ['hello', 'world']]
>>> qt = QTable(dat, names=['a', 'b'], units=(u.m, None))
>>> qt = QTable(dat, names=['a', 'b'], units={'a': u.m})
```

descriptions
This allows for setting the description for one or more columns at the time of creating the table. The input can be either a list of description values corresponding to each of the columns in the table (using `None` for no description), or a `dict` that provides the description for specified column names. This works in the same way as the `units` example above.

Copy versus Reference

Normally when a new **`Table`** object is created, the input data are *copied* into a new internal array object. This ensures that if the new table elements are modified then the original data will not be affected. However, when creating a table from a `numpy` ndarray object (structured or homogeneous) or a dict, it is possible to disable copying so that a memory reference to the original data is used instead. This has the advantage of being faster and using less memory. However, caution must be exercised because the new table data and original data will be linked, as shown below:

```
>>> arr = np.array([(1, 2.0, 'x'),
...                  (4, 5.0, 'y')],
...                 dtype=[('a', 'i8'), ('b', 'f8'), ('c', 'S2')])
>>> print(arr['a'])  # column "a" of the input array
[1 4]
>>> t = Table(arr, copy=False)
>>> t['a'][1] = 99
>>> print(arr['a'])  # arr['a'] got changed when we modified t['a']
[ 1 99]
```

Note that when referencing the data it is not possible to change the data types since that operation requires making a copy of the data. In this case an error occurs:

```
>>> t = Table(arr, copy=False, dtype=('f4', 'i4', 'S4'))
Traceback (most recent call last):
  ...
ValueError: Cannot specify dtype when copy=False
```

Another caveat to using referenced data is that if you add a new row to the table, the reference to the original data array is lost and the table will now instead hold a copy of the original values (in addition to the new row).

Column and TableColumns Classes

There are two classes, `Column` and `TableColumns`, that are useful when constructing new tables.

Column

A `Column` object can be created as follows, where in all cases the column `name` should be provided as a keyword argument and you can optionally provide these values:

`data` : *list, ndarray or None*
 Column data values.

`dtype` : *numpy.dtype compatible value*
 Data type for column.

`description` : *str*
 Full description of column.

`unit` : *str*
 Physical unit.

`format` : *str or function*
 Format specifier for outputting column values.

`meta` : *dict*
 Metadata associated with the column.

Initialization Options

The column data values, shape, and data type are specified in one of two ways:

Provide a ``data`` value but not a ``length`` or ``shape``

Examples:

```
col = Column([1, 2], name='a')  # shape=(2,)
col = Column([[1, 2], [3, 4]], name='a')  # shape=(2, 2)
col = Column([1, 2], name='a', dtype=float)
col = Column(np.array([1, 2]), name='a')
col = Column(['hello', 'world'], name='a')
```

The `dtype` argument can be any value which is an acceptable fixed-size data type initializer for the `numpy.dtype()` method. See the reference for data type objects. Examples include:

- Python non-string type (float, int, bool).
- `numpy` non-string type (e.g., np.float32, np.int64).
- `numpy.dtype` array-protocol type strings (e.g., 'i4', 'f8', 'S15').

If no `dtype` value is provided, then the type is inferred using `np.array(data)`. When `data` is provided then the `shape` and `length` arguments are ignored.

Provide ``length`` and optionally ``shape``, but not ``data``

Examples:

```
col = Column(name='a', length=5)
col = Column(name='a', dtype=int, length=10, shape=(3,4))
```

The default `dtype` is `np.float64`. The `shape` argument is the array shape of a single cell in the column. The default `shape` is () which means a single value in each element.

Note

After setting the type for a column, that type cannot be changed. If data values of a different type are assigned to the column then they will be cast to the existing column type.

Format Specifier

The format specifier controls the output of column values when a table or column is printed or written to an ASCII table. In the simplest case, it is a string that can be passed to Python's built-in format function. For more complicated formatting, one can also give "old style" or "new style" format strings, or even a function:

Plain format specification

This type of string specifies directly how the value should be formatted using a format specification mini-language that is quite similar to C.

`".4f"` will give four digits after the decimal in float format, or

`"6d"` will give integers in six-character fields.

Old style format string

This corresponds to syntax like `"%.4f" % value` as documented in printf-style String Formatting.

> `"%.4f"` to print four digits after the decimal in float format, or
>
> `"%6d"` to print an integer in a six-character wide field.

New style format string

This corresponds to syntax like `"{:.4f}".format(value)` as documented in format string syntax.

> `"{:.4f}"` to print four digits after the decimal in float format, or
>
> `"{:6d}"` to print an integer in a six-character wide field.

Note that in either format string case any Python string that formats exactly one value is valid, so `{:.4f} angstroms` or `Value: %12.2f` would both work.

Function

The greatest flexibility can be achieved by setting a formatting function. This function must accept a single argument (the value) and return a string. In the following example this is used to make a LaTeX ready output:

```
>>> t = Table([[1,2],[1.234e9,2.34e-12]], names = ('a','b'))
>>> def latex_exp(value):
...     val = '{0:8.2}'.format(value)
...     mant, exp = val.split('e')
...     # remove leading zeros
...     exp = exp[0] + exp[1:].lstrip('0')
...     return '$ {0} \\times 10^{{ {1} }}$' .format(mant, exp)
>>> t['b'].format = latex_exp
>>> t['a'].format = '.4f'
>>> import sys
>>> t.write(sys.stdout, format='latex')
\begin{table}
\begin{tabular}{cc}
a & b \\
1.0000 & $  1.2 \times 10^{ +9 }$ \\
2.0000 & $  2.3 \times 10^{ -12 }$ \\
\end{tabular}
\end{table}
```

TableColumns

Each **Table** object has an attribute `columns` which is an ordered dictionary that stores all of the **Column** objects in the table (see also the Column section). Technically, the `columns` attribute is a **TableColumns** object, which is an enhanced ordered dictionary that provides easier ways to select multiple columns. There are a few key points to remember:

- A **Table** can be initialized from a **TableColumns** object (copy is always True).
- Selecting multiple columns from a **TableColumns** object returns another **TableColumns** object.
- Select one column from a **TableColumns** object returns a **Column**.

So now look at the ways to select columns from a **TableColumns** object:

Select columns by name

```
>>> t = Table(names=('a', 'b', 'c', 'd'))

>>> t.columns['d', 'c', 'b']
<TableColumns names=('d','c','b')>
```

Select columns by index slicing

```
>>> t.columns[0:2]  # Select first two columns
<TableColumns names=('a','b')>

>>> t.columns[::-1]  # Reverse column order
<TableColumns names=('d','c','b','a')>
```

Select columns by index or name

```
>>> t.columns[1]  # Choose columns by index
<Column name='b' dtype='float64' length=0>

>>> t.columns['b']  # Choose column by name
<Column name='b' dtype='float64' length=0>
```

Subclassing Table

For some applications it can be useful to subclass the **Table** class in order to introduce specialized behavior. Here we address two particular use cases for subclassing: adding custom table attributes and changing the behavior of internal class objects.

Adding Custom Table Attributes

One simple customization that can be useful is adding new attributes to the table object. There is nothing preventing setting an attribute on an existing table object, for example `t.foo = 'hello'`. However, this attribute would be ephemeral because it will be lost if the table is sliced, copied, or pickled. Instead, you can add persistent attributes as shown in this example:

```
from astropy.table import Table, TableAttribute

class MyTable(Table):
    foo = TableAttribute()
    bar = TableAttribute(default=[])
    baz = TableAttribute(default=1)

t = MyTable([[1, 2]], foo='foo')
t.bar.append(2.0)
t.baz = 'baz'
```

Some key points:

- A custom attribute can be set when the table is created or using the usual syntax for setting an object attribute.
- A custom attribute always has a default value, either explicitly set in the class definition or `None`.
- The attribute values are stored in the table `meta` dictionary. This is the mechanism by which they are persistent through copy, slice, and serialization such as pickling or writing to an ECSV ASCII file.

Changing Behavior of Internal Class Objects

It is also possible to change the behavior of the internal class objects which are contained or created by a Table. This includes rows, columns, formatting, and the columns container. In order to do this the subclass needs to declare what class to use (if it is different from the built-in version). This is done by specifying one or more of the class attributes `Row`, `Column`, `MaskedColumn`, `TableColumns`, or `TableFormatter`.

The following trivial example overrides all of these with do-nothing subclasses, but in practice you would override only the necessary subcomponents:

```
>>> from astropy.table import Table, Row, Column, MaskedColumn,
TableColumns, TableFormatter

>>> class MyRow(Row): pass
>>> class MyColumn(Column): pass
>>> class MyMaskedColumn(MaskedColumn): pass
>>> class MyTableColumns(TableColumns): pass
>>> class MyTableFormatter(TableFormatter): pass

>>> class MyTable(Table):
...     """
...     Custom subclass of astropy.table.Table
...     """
...     Row = MyRow  # Use MyRow to create a row object
...     Column = MyColumn  # Column
...     MaskedColumn = MyMaskedColumn  # Masked Column
...     TableColumns = MyTableColumns  # Ordered dict holding Column
objects
...     TableFormatter = MyTableFormatter  # Controls table output
```

Example

As a more practical example, suppose you have a table of data with a certain set of fixed columns, but you also want to carry an arbitrary dictionary of keyword=value parameters for each row and then access those values using the same item access syntax as if they were columns. It is assumed here that the extra parameters are contained in a `numpy` object-dtype column named `params`:

```
>>> from astropy.table import Table, Row
>>> class ParamsRow(Row):
...    """
...    Row class that allows access to an arbitrary dict of
parameters
...    stored as a dict object in the ``params`` column.
...    """
...    def __getitem__(self, item):
...        if item not in self.colnames:
...            return super().__getitem__('params')[item]
...        else:
...            return super().__getitem__(item)
...
...    def keys(self):
...        out = [name for name in self.colnames if name != 'params']
...        params = [key.lower() for key in sorted(self['params'])]
...        return out + params
...
...    def values(self):
...        return [self[key] for key in self.keys()]
```

Now we put this into action with a trivial **`Table`** subclass:

```
>>> class ParamsTable(Table):
...     Row = ParamsRow
```

First make a table and add a couple of rows:

```
>>> t = ParamsTable(names=['a', 'b', 'params'], dtype=['i', 'f',
'O'])
```

```
>>> t.add_row((1, 2.0, {'x': 1.5, 'y': 2.5}))
>>> t.add_row((2, 3.0, {'z': 'hello', 'id': 123123}))
>>> print(t)
 a   b              params
--- --- ----------------------------
  1 2.0         {'y': 2.5, 'x': 1.5}
  2 3.0 {'z': 'hello', 'id': 123123}
```

Now see what we have from our specialized `ParamsRow` object:

```
>>> t[0]['y']
2.5
>>> t[1]['id']
123123
>>> t[1].keys()
['a', 'b', 'id', 'z']
>>> t[1].values()
[2, 3.0, 123123, 'hello']
```

To make this example really useful, you might want to override `Table.__getitem__` in order to allow table-level access to the parameter fields. This might look something like:

```
class ParamsTable(table.Table):
    Row = ParamsRow

    def __getitem__(self, item):
        if isinstance(item, str):
            if item in self.colnames:
                return self.columns[item]
            else:
                # If item is not a column name then create a new
MaskedArray
                # corresponding to self['params'][item] for each row.
This
                # might not exist in some rows so mark as masked
(missing) in
                # those cases.
                mask = np.zeros(len(self), dtype=np.bool_)
                item = item.upper()
                values = [params.get(item) for params in
self['params']]
                for ii, value in enumerate(values):
                    if value is None:
                        mask[ii] = True
                        values[ii] = ''
                return self.MaskedColumn(name=item, data=values,
mask=mask)

        # ... and then the rest of the original __getitem__ ...
```

Columns and Quantities

`astropy` **Quantity** objects can be handled within tables in two complementary ways. The first method stores the **Quantity** object natively within the table via the “mixin” column protocol. See the sections on Mixin Columns and Quantity and QTable for details, but in brief, the key difference is using the **QTable** class to indicate that a **Quantity** should be stored natively within the table:

```
>>> from astropy.table import QTable
>>> from astropy import units as u
>>> t = QTable()
>>> t['velocity'] = [3, 4] * u.m / u.s
>>> type(t['velocity'])
astropy.units.quantity.Quantity
```

For new code that is quantity-aware we recommend using **QTable**, but this may not be possible in all situations (particularly when interfacing with legacy code that does not handle quantities) and there are Details and Caveats that apply. In this case, use the **Table** class, which will convert a **Quantity** to a **Column** object with a `unit` attribute:

```
>>> from astropy.table import Table
>>> t = Table()
>>> t['velocity'] = [3, 4] * u.m / u.s
>>> type(t['velocity'])
astropy.table.column.Column
>>> t['velocity'].unit
Unit("m / s")
```

To learn more about using standard **Column** objects with defined units, see the Columns with Units section.

Table-Like Objects

In order to improve interoperability between different table classes, an `astropy` **Table** object can be created directly from any other table-like object that provides an `__astropy_table__` method. In this case the `__astropy_table__` method will be called as follows:

```
>>> data = SomeOtherTableClass({'a': [1, 2], 'b': [3, 4]})
>>> t = QTable(data, copy=False, strict_copy=True)
```

Internally the following call will be made to ask the `data` object to return a representation of itself as an `astropy` **Table**, respecting the `copy` preference of the original call to `QTable()`:

```
data.__astropy_table__(cls, copy, **kwargs)
```

Here `cls` is the **Table** class or subclass that is being instantiated (**QTable** in this example), `copy` indicates whether a copy of the values in `data` should be provided, and `**kwargs` are any extra keyword arguments which are not valid **Table** `_init_()` keyword arguments. In the example above, `strict_copy=True` would end up in `**kwargs` and get passed to `__astropy_table__()`.

If `copy` is `True` then the `__astropy_table__` method must ensure that a copy of the original data is returned. If `copy` is `False` then a reference to the table data should returned if possible. If it is not possible (e.g., the original data are in a Python list or must be otherwise transformed in memory) then `__astropy_table__` method is free to either return a copy or else raise an exception. This choice depends on the preference of the implementation. The implementation might choose to allow an additional keyword argument (e.g., `strict_copy` which gets passed via `**kwargs`) to control the behavior in this case.

As a concise example, imagine a dict-based table class. (Note that **Table** already can be initialized from a dict-like object, so this is a bit contrived but does illustrate the principles involved.) Please pay attention to the method signature:

```
def __astropy_table__(self, cls, copy, **kwargs):
```

Your class implementation of this must use the `**kwargs` technique for catching keyword arguments at the end. This is to ensure future compatibility in case additional keywords are added to the internal `table = data.__astropy_table__(cls, copy)` call. Including `**kwargs` will prevent breakage in this case.

```
class DictTable(dict):
    """
    Trivial "table" class that just uses a dict to hold columns.
    This does not actually implement anything useful that makes
    this a table.

    The non-standard ``strict_copy=False`` keyword arg here will be
passed
```

```
    via the **kwargs of Table __init__().
    """

    def __astropy_table__(self, cls, copy, strict_copy=False,
**kwargs):
        """
        Return an astropy Table of type ``cls``.

        Parameters
        ----------
        cls : type
             Astropy ``Table`` class or subclass.
        copy : bool
             Copy input data (True) or return a reference (False).
        strict_copy : bool, optional
             Raise an exception if copy is False but reference is not
             possible.
        **kwargs : dict, optional
             Additional keyword args (ignored currently).
        """
        if kwargs:
            warnings.warn('unexpected keyword args
{}'.format(kwargs))

        cols = list(self.values())
        names = list(self.keys())

        # If returning a reference to existing data (copy=False) and
        # strict_copy=True, make sure that each column is a numpy
ndarray.
        # If a column is a Python list or tuple then it must be
copied for
        # representation in an astropy Table.

        if not copy and strict_copy:
            for name, col in zip(names, cols):
                if not isinstance(col, np.ndarray):
                    raise ValueError('cannot have copy=False because
column {} is '
                                     'not an ndarray'.format(name))

        return cls(cols, names=names, copy=copy)
```

Access Table

Accessing a Table

Accessing table properties and data is generally consistent with the basic interface for `numpy` structured arrays.

Basics

For a quick overview, the code below shows the basics of accessing table data. Where relevant, there is a comment about what sort of object is returned. Except where noted, table access returns objects that can be modified in order to update the original table data or properties. See also the section on Copy versus Reference to learn more about this topic.

Make table

```
from astropy.table import Table
import numpy as np

arr = np.arange(15).reshape(5, 3)
t = Table(arr, names=('a', 'b', 'c'), meta={'keywords': {'key1':
'val1'}})
```

Table properties

```
t.columns   # Dict of table columns (access by column name, index, or
slice)
t.colnames  # List of column names
t.meta      # Dict of meta-data
len(t)      # Number of table rows
```

Access table data

```
t['a']       # Column 'a'
t['a'][1]    # Row 1 of column 'a'
t[1]         # Row obj for with row 1 values
t[1]['a']    # Column 'a' of row 1
t[2:5]       # Table object with rows 2:5
t[[1, 3, 4]]  # Table object with rows 1, 3, 4 (copy)
t[np.array([1, 3, 4])]  # Table object with rows 1, 3, 4 (copy)
t[[]]        # Same table definition but with no rows of data
t['a', 'c']  # Table with cols 'a', 'c' (copy)
dat = np.array(t)  # Copy table data to numpy structured array object
t['a'].quantity  # an astropy.units.Quantity for Column 'a'
t['a'].to('km')  # an astropy.units.Quantity for Column 'a' in units
of kilometers
t.columns[1]  # Column 1 (which is the 'b' column)
t.columns[0:2]  # New table with columns 0 and 1
```

Note

Although they appear nearly equivalent, there is a factor of two performance difference between `t[1]['a']` (slower, because an intermediate **Row** object gets created) versus `t['a'][1]` (faster). Always use the latter when possible.

Print table or column

```
print(t)     # Print formatted version of table to the screen
t.pprint()   # Same as above
t.pprint(show_unit=True)  # Show column unit
t.pprint(show_name=False)  # Do not show column names
t.pprint_all() # Print full table no matter how long / wide it is
(same as t.pprint(max_lines=-1, max_width=-1))

t.more()  # Interactively scroll through table like Unix "more"

print(t['a'])    # Formatted column values
t['a'].pprint()  # Same as above, with same options as Table.pprint()
t['a'].more()    # Interactively scroll through column

lines = t.pformat()  # Formatted table as a list of lines (same
options as pprint)
lines = t['a'].pformat()  # Formatted column values as a list
```

Details

For all of the following examples it is assumed that the table has been created as follows:

```
>>> from astropy.table import Table, Column
>>> import numpy as np
>>> import astropy.units as u

>>> arr = np.arange(15, dtype=np.int32).reshape(5, 3)
>>> t = Table(arr, names=('a', 'b', 'c'), meta={'keywords': {'key1':
'val1'}})
>>> t['a'].format = "%6.3f"  # print as a float with 3 digits after
decimal point
>>> t['a'].unit = 'm sec^-1'
>>> t['a'].description = 'unladen swallow velocity'
>>> print(t)
     a      b   c
  m sec^-1
  -------- --- ---
     0.000   1   2
     3.000   4   5
     6.000   7   8
     9.000  10  11
    12.000  13  14
```

Note

In the example above the `format`, `unit`, and `description` attributes of the **`Column`** were set directly. For Mixin Columns like **`Quantity`** you must set via the `info` attribute, for example, `t['a'].info.format = "%6.3f"`. You can use the `info` attribute with **`Column`** objects as well, so the general solution that works with any table column is to set via the `info` attribute. See Mixin Attributes for more information.

Summary Information

You can get summary information about the table as follows:

```
>>> t.info
<Table length=5>
name dtype   unit   format       description
---- ----- -------- ------ ------------------------
   a int32 m sec^-1  %6.3f unladen swallow velocity
   b int32
   c int32
```

If called as a function then you can supply an `option` that specifies the type of information to return. The built-in `option` choices are `attributes` (column attributes, which is the default) or `stats` (basic column statistics). The `option` argument can also be a list of available options:

```
>>> t.info('stats')
<Table length=5>
name mean      std      min max
---- ---- ------------- --- ---
   a  6.0 4.24264068712   0  12
   b  7.0 4.24264068712   1  13
   c  8.0 4.24264068712   2  14

>>> t.info(['attributes', 'stats'])
<Table length=5>
name dtype   unit   format       description        mean      std
min max
---- ----- -------- ------ ------------------------ ----
------------- --- ---
   a int32 m sec^-1  %6.3f unladen swallow velocity  6.0
4.24264068712   0  12
   b int32                                           7.0
```

```
4.24264068712   1  13
   c int32                                            8.0
4.24264068712   2  14
```

Columns also have an `info` property that has the behavior and arguments, but provides information about a single column:

```
>>> t['a'].info
name = a
dtype = int32
unit = m sec^-1
format = %6.3f
description = unladen swallow velocity
class = Column
n_bad = 0
length = 5

>>> t['a'].info('stats')
name = a
mean = 6.0
std = 4.24264068712
min = 0
max = 12
n_bad = 0
length = 5
```

Accessing Properties

The code below shows accessing the table columns as a **`TableColumns`** object, getting the column names, table metadata, and number of table rows. The table metadata is an ordered dictionary (OrderedDict) by default.

```
>>> t.columns
<TableColumns names=('a','b','c')>

>>> t.colnames
['a', 'b', 'c']

>>> t.meta  # Dict of meta-data
{'keywords': {'key1': 'val1'}}

>>> len(t)
5
```

Accessing Data

As expected you can access a table column by name and get an element from that column with a numerical index:

```
>>> t['a']  # Column 'a'
<Column name='a' dtype='int32' unit='m sec^-1' format='%6.3f'
description='unladen swallow velocity' length=5>
 0.000
 3.000
 6.000
 9.000
12.000


>>> t['a'][1]  # Row 1 of column 'a'
3
```

When a table column is printed, it is formatted according to the `format` attribute (see Format Specifier). Note the difference between the column representation above and how it appears via `print()` or `str()`:

```
>>> print(t['a'])
   a
m sec^-1
--------
   0.000
   3.000
   6.000
   9.000
  12.000
```

Likewise a table row and a column from that row can be selected:

```
>>> t[1]  # Row object corresponding to row 1
<Row index=1>
   a       b     c
m sec^-1
 int32   int32 int32
-------- ----- -----
   3.000     4     5

>>> t[1]['a']  # Column 'a' of row 1
3
```

A **`Row`** object has the same columns and metadata as its parent table:

```
>>> t[1].columns
<TableColumns names=('a','b','c')>

>>> t[1].colnames
```

```
['a', 'b', 'c']
```

Slicing a table returns a new table object with references to the original data within the slice region (See Copy versus Reference). The table metadata and column definitions are copied.

```
>>> t[2:5]  # Table object with rows 2:5 (reference)
<Table length=3>
   a       b     c
m sec^-1
 int32   int32 int32
-------- ----- -----
   6.000     7     8
   9.000    10    11
  12.000    13    14
```

It is possible to select table rows with an array of indexes or by specifying multiple column names. This returns a copy of the original table for the selected rows or columns.

```
>>> print(t[[1, 3, 4]])  # Table object with rows 1, 3, 4 (copy)
     a      b   c
  m sec^-1
  -------- --- ---
     3.000   4   5
     9.000  10  11
    12.000  13  14


>>> print(t[np.array([1, 3, 4])])  # Table object with rows 1, 3, 4
(copy)
     a      b   c
  m sec^-1
  -------- --- ---
     3.000   4   5
     9.000  10  11
    12.000  13  14


>>> print(t['a', 'c'])  # or t[['a', 'c']] or t[('a', 'c')]
...                     # Table with cols 'a', 'c' (copy)
     a      c
  m sec^-1
  -------- ---
     0.000   2
     3.000   5
     6.000   8
    9.000  11
   12.000  14
```

We can select rows from a table using conditionals to create boolean masks. A table indexed with a boolean array will only return rows where the mask array element is **True**. Different conditionals can be combined using the bitwise operators.

```
>>> mask = (t['a'] > 4) & (t['b'] > 8)  # Table rows where column a >
4
>>> print(t[mask])                      # and b > 8
...
     a       b   c
  m sec^-1
  -------- --- ---
     9.000  10  11
    12.000  13  14
```

Finally, you can access the underlying table data as a native `numpy` structured array by creating a copy or reference with `np.array`:

```
>>> data = np.array(t)  # copy of data in t as a structured array
>>> data = np.array(t, copy=False)  # reference to data in t
```

Table Equality

We can check table data equality using two different methods:

- The `==` comparison operator. This returns a `True` or `False` for each row if the *entire row* matches. This is the same as the behavior of `numpy` structured arrays.
- Table **values_equal()** to compare table values element-wise. This returns a boolean `True` or `False` for each table *element*, so you get a **Table** of values.

Examples

To check table equality:

```
>>> t1 = Table(rows=[[1, 2, 3],
...                  [4, 5, 6],
...                  [7, 7, 9]], names=['a', 'b', 'c'])
>>> t2 = Table(rows=[[1, 2, -1],
...                  [4, -1, 6],
...                  [7, 7, 9]], names=['a', 'b', 'c'])

>>> t1 == t2
array([False, False,  True])
```

```
>>> t1.values_equal(t2)  # Compare to another table
<Table length=3>
 a     b     c
bool  bool  bool
---- ----- -----
True  True False
True False  True
True  True  True

>>> t1.values_equal([2, 4, 7])  # Compare to an array column-wise
<Table length=3>
  a     b     c
 bool  bool  bool
----- ----- -----
False  True False
 True False False
 True  True False

>>> t1.values_equal(7)  # Compare to a scalar column-wise
<Table length=3>
  a     b     c
 bool  bool  bool
----- ----- -----
False False False
False False False
 True  True False
```

Formatted Printing

The values in a table or column can be printed or retrieved as a formatted table using one of several methods:

- `print()` function.
- Table `more()` or Column `more()` methods to interactively scroll through table values.
- Table `pprint()` or Column `pprint()` methods to print a formatted version of the table to the screen.
- Table `pformat()` or Column `pformat()` methods to return the formatted table or column as a list of fixed-width strings. This could be used as a quick way to save a table.

These methods use Format Specifier if available and strive to make the output readable. By default, table and column printing will not print the table larger than the available interactive screen size. If the screen size cannot be determined (in a non-interactive environment or on Windows) then a default size of 25 rows by 80 columns is used. If a table is too large, then rows and/or columns are cut from the middle so it fits.

Example

To print a formatted table:

```
>>> arr = np.arange(3000).reshape(100, 30)  # 100 rows x 30 columns
array
>>> t = Table(arr)
>>> print(t)
col0 col1 col2 col3 col4 col5 col6 ... col23 col24 col25 col26 col27
col28 col29
---- ---- ---- ---- ---- ---- ---- ... ----- ----- ----- ----- -----
----- -----
   0    1    2    3    4    5    6 ...    23    24    25    26    27
28    29
  30   31   32   33   34   35   36 ...    53    54    55    56    57
58    59
  60   61   62   63   64   65   66 ...    83    84    85    86    87
88    89
  90   91   92   93   94   95   96 ...   113   114   115   116   117
118   119
 120  121  122  123  124  125  126 ...   143   144   145   146   147
148   149
 150  151  152  153  154  155  156 ...   173   174   175   176   177
178   179
 180  181  182  183  184  185  186 ...   203   204   205   206   207
208   209
 210  211  212  213  214  215  216 ...   233   234   235   236   237
238   239
 240  241  242  243  244  245  246 ...   263   264   265   266   267
268   269
 270  271  272  273  274  275  276 ...   293   294   295   296   297
298   299
 ...  ...  ...  ...  ...  ...  ... ...   ...   ...   ...   ...   ...
...   ...
2670 2671 2672 2673 2674 2675 2676 ...  2693  2694  2695  2696  2697
2698  2699
2700 2701 2702 2703 2704 2705 2706 ...  2723  2724  2725  2726  2727
2728  2729
2730 2731 2732 2733 2734 2735 2736 ...  2753  2754  2755  2756  2757
2758  2759
2760 2761 2762 2763 2764 2765 2766 ...  2783  2784  2785  2786  2787
2788  2789
2790 2791 2792 2793 2794 2795 2796 ...  2813  2814  2815  2816  2817
2818  2819
2820 2821 2822 2823 2824 2825 2826 ...  2843  2844  2845  2846  2847
2848  2849
2850 2851 2852 2853 2854 2855 2856 ...  2873  2874  2875  2876  2877
2878  2879
2880 2881 2882 2883 2884 2885 2886 ...  2903  2904  2905  2906  2907
```

```
2908  2909
2910 2911 2912 2913 2914 2915 2916 ...  2933  2934  2935  2936  2937
2938  2939
2940 2941 2942 2943 2944 2945 2946 ...  2963  2964  2965  2966  2967
2968  2969
2970 2971 2972 2973 2974 2975 2976 ...  2993  2994  2995  2996  2997
2998  2999
Length = 100 rows
```

more() method

In order to browse all rows of a table or column use the Table **`more()`** or Column **`more()`** methods. These let you interactively scroll through the rows much like the Linux `more` command. Once part of the table or column is displayed the supported navigation keys are:

f, space : forward one page
b : back one page
r : refresh same page
n : next row
p : previous row
< : go to beginning
> : go to end
q : quit browsing
h : print this help

pprint() method

In order to fully control the print output use the Table **`pprint()`** or Column **`pprint()`** methods. These have keyword arguments `max_lines`, `max_width`, `show_name`, `show_unit` with meanings as shown below:

```
>>> arr = np.arange(3000, dtype=float).reshape(100, 30)
>>> t = Table(arr)
>>> t['col0'].format = '%e'
>>> t['col1'].format = '%.6f'
>>> t['col0'].unit = 'km**2'
>>> t['col29'].unit = 'kg sec m**-2'

>>> t.pprint(max_lines=8, max_width=40)
    col0     ...    col29
    km2      ... kg sec m**-2
------------ ... ------------
0.000000e+00 ...         29.0
         ... ...          ...
2.940000e+03 ...       2969.0
2.970000e+03 ...       2999.0
Length = 100 rows
```

```
>>> t.pprint(max_lines=8, max_width=40, show_unit=True)
    col0      ...    col29
    km2       ... kg sec m**-2
------------ ... ------------
0.000000e+00 ...         29.0
         ... ...          ...
2.940000e+03 ...       2969.0
2.970000e+03 ...       2999.0
Length = 100 rows

>>> t.pprint(max_lines=8, max_width=40, show_name=False)
    km2       ... kg sec m**-2
------------ ... ------------
0.000000e+00 ...         29.0
3.000000e+01 ...         59.0
         ... ...          ...
2.940000e+03 ...       2969.0
2.970000e+03 ...       2999.0
Length = 100 rows
```

In order to force printing all values regardless of the output length or width use **pprint_all()**, which is equivalent to setting `max_lines` and `max_width` to `-1` in **pprint()**. **pprint_all()** takes the same arguments as **pprint()**. For the wide table in this example you see six lines of wrapped output like the following:

```
>>> t.pprint_all(max_lines=8)
    col0         col1     col2   col3   col4   col5   col6   col7
col8   col9  col10  col11  col12  col13  col14  col15  col16  col17
col18  col19  col20  col21  col22  col23  col24  col25  col26  col27
col28     col29
    km2
kg sec m**-2
------------ ----------- ------ ------ ------ ------ ------ ------
------ ------ ------ ------ ------ ------ ------ ------ ------ ------
------ ------ ------ ------ ------ ------ ------ ------ ------ ------
------ ------------
0.000000e+00    1.000000    2.0    3.0    4.0    5.0    6.0    7.0
8.0    9.0   10.0   11.0   12.0   13.0   14.0   15.0   16.0   17.0
18.0   19.0   20.0   21.0   22.0   23.0   24.0   25.0   26.0   27.0
28.0          29.0
         ...         ...    ...    ...    ...    ...    ...    ...
...    ...    ...    ...    ...    ...    ...    ...    ...    ...
...    ...    ...    ...    ...    ...    ...    ...    ...    ...
...          ...
2.940000e+03 2941.000000 2942.0 2943.0 2944.0 2945.0 2946.0 2947.0
```

```
2948.0 2949.0 2950.0 2951.0 2952.0 2953.0 2954.0 2955.0 2956.0 2957.0
2958.0 2959.0 2960.0 2961.0 2962.0 2963.0 2964.0 2965.0 2966.0 2967.0
2968.0       2969.0
2.970000e+03 2971.000000 2972.0 2973.0 2974.0 2975.0 2976.0 2977.0
2978.0 2979.0 2980.0 2981.0 2982.0 2983.0 2984.0 2985.0 2986.0 2987.0
2988.0 2989.0 2990.0 2991.0 2992.0 2993.0 2994.0 2995.0 2996.0 2997.0
2998.0       2999.0
Length = 100 rows
```

For columns, the syntax and behavior of **`pprint()`** is the same except that there is no `max_width` keyword argument:

```
>>> t['col3'].pprint(max_lines=8)
 col3
------
   3.0
  33.0
   ...
2943.0
2973.0
Length = 100 rows
```

Column alignment

Individual columns have the ability to be aligned in a number of different ways for an enhanced viewing experience:

```
>>> t1 = Table()
>>> t1['long column name 1'] = [1, 2, 3]
>>> t1['long column name 2'] = [4, 5, 6]
>>> t1['long column name 3'] = [7, 8, 9]
>>> t1['long column name 4'] = [700000, 800000, 900000]
>>> t1['long column name 2'].info.format = '<'
>>> t1['long column name 3'].info.format = '0='
>>> t1['long column name 4'].info.format = '^'
>>> t1.pprint()
 long column name 1 long column name 2 long column name 3 long column
name 4
------------------ ------------------ ------------------
------------------
                 1 4                  000000000000000007       700000
                 2 5                  000000000000000008       800000
                 3 6                  000000000000000009       900000
```

Conveniently, alignment can be handled another way — by passing a list to the keyword argument `align`:

```
>>> t1 = Table()
>>> t1['column1'] = [1, 2, 3]
>>> t1['column2'] = [2, 4, 6]
>>> t1.pprint(align=['<', '0='])
column1 column2
------- -------
1       0000002
2       0000004
3       0000006
```

It is also possible to set the alignment of all columns with a single string value:

```
>>> t1.pprint(align='^')
column1 column2
------- -------
   1       2
   2       4
   3       6
```

The fill character for justification can be set as a prefix to the alignment character (see Format Specification Mini-Language for additional explanation). This can be done both in the `align` argument and in the column `format` attribute. Note the interesting interaction below:

```
>>> t1 = Table([[1.0, 2.0], [1, 2]], names=['column1', 'column2'])

>>> t1['column1'].format = '#^.2f'
>>> t1.pprint()
column1 column2
------- -------
##1.00#       1
##2.00#       2
```

Now if we set a global align, it seems like our original column format got lost:

```
>>> t1.pprint(align='!<')
column1 column2
------- -------
1.00!!! 1!!!!!!
2.00!!! 2!!!!!!
```

The way to avoid this is to explicitly specify the alignment strings for every column and use `None` where the column format should be used:

```
>>> t1.pprint(align=[None, '!<'])
column1 column2
```

```
------- -------
##1.00# 1!!!!!!
##2.00# 2!!!!!!
```

pformat() method

In order to get the formatted output for manipulation or writing to a file use the Table `pformat()` or Column `pformat()` methods. These behave just as for `pprint()` but return a list corresponding to each formatted line in the `pprint()` output. The `pformat_all()` method can be used to return a list for all lines in the Table.

```
>>> lines = t['col3'].pformat(max_lines=8)
```

Multidimensional columns

If a column has more than one dimension then each element of the column is itself an array. In the example below there are three rows, each of which is a `2 x 2` array. The formatted output for such a column shows only the first and last value of each row element and indicates the array dimensions in the column name header:

```
>>> from astropy.table import Table, Column
>>> import numpy as np
>>> t = Table()
>>> arr = [ np.array([[ 1,  2],
...                   [10, 20]]),
...         np.array([[ 3,  4],
...                   [30, 40]]),
...         np.array([[ 5,  6],
...                   [50, 60]]) ]
>>> t['a'] = arr
>>> t['a'].shape
(3, 2, 2)
>>> t.pprint()
a [2,2]
-------
1 .. 20
3 .. 40
5 .. 60
```

In order to see all of the data values for a multidimensional column use the column representation. This uses the standard `numpy` mechanism for printing any array:

```
>>> t['a'].data
array([[[ 1,  2],
        [10, 20]],
       [[ 3,  4],
        [30, 40]],
       [[ 5,  6],
        [50, 60]]])
```

Columns with Units

A **Column** object with units within a standard **Table** (as opposed to a **QTable**) has certain quantity-related conveniences available. To begin with, it can be converted explicitly to a **Quantity** object via the **quantity** property and the **to()** method:

```
>>> data = [[1., 2., 3.], [40000., 50000., 60000.]]
>>> t = Table(data, names=('a', 'b'))
>>> t['a'].unit = u.m
>>> t['b'].unit = 'km/s'
>>> t['a'].quantity
<Quantity [1., 2., 3.] m>
>>> t['b'].to(u.kpc/u.Myr)
<Quantity [40.9084866 , 51.13560825, 61.3627299 ] kpc / Myr>
```

Note that the **quantity** property is actually a *view* of the data in the column, not a copy. Hence, you can set the values of a column in a way that respects units by making in-place changes to the **quantity** property:

```
>>> t['b']
<Column name='b' dtype='float64' unit='km / s' length=3>
40000.0
50000.0
60000.0

>>> t['b'].quantity[0] = 45000000*u.m/u.s
>>> t['b']
<Column name='b' dtype='float64' unit='km / s' length=3>
45000.0
50000.0
60000.0
```

Even without explicit conversion, columns with units can be treated like like an `astropy` **Quantity** in *some* arithmetic expressions (see the warning below for caveats to this):

```
>>> t['a'] + .005*u.km
<Quantity [6., 7., 8.] m>
>>> from astropy.constants import c
>>> (t['b'] / c).decompose()
```

```
<Quantity [0.15010384, 0.16678205, 0.20013846]>
```

Warning

Table columns do *not* always behave the same as **Quantity**. Table columns act more like regular `numpy` arrays unless either explicitly converted to a **Quantity** or combined with an **Quantity** using an arithmetic operator. For example, the following does not work in the way you would expect:

```
>>> import numpy as np
>>> from astropy.table import Table
>>> data = [[30, 90]]
>>> t = Table(data, names=('angle',))
>>> t['angle'].unit = 'deg'
>>> np.sin(t['angle'])
<Column name='angle' dtype='float64' unit='deg' length=2>
-0.988031624093
 0.893996663601
```

This is wrong both in that it says the unit is degrees, *and* `sin` treated the values and radians rather than degrees. If at all in doubt that you will get the right result, the safest choice is to either use **QTable** or to explicitly convert to **Quantity**:

```
>>> np.sin(t['angle'].quantity)
<Quantity [0.5, 1. ]>
```

Bytestring Columns

Using bytestring columns (`numpy` `'S'` dtype) is possible with `astropy` tables since they can be compared with the natural Python string (`str`) type. See The bytes/str dichotomy in Python 3 for a very brief overview of the difference.

The standard method of representing strings in `numpy` is via the unicode `'U'` dtype. The problem is that this requires 4 bytes per character, and if you have a very large number of strings in memory this could fill memory and impact performance. A very common use case is that these strings are actually ASCII and can be represented with 1 byte per character. In `astropy` it is possible to work directly and conveniently with bytestring data in **Table** and **Column** operations.

Note that the bytestring issue is a particular problem when dealing with HDF5 files, where character data are read as bytestrings (`'S'` dtype) when using the Unified File Read/Write Interface. Since HDF5 files are frequently used to store very large datasets, the memory bloat associated with conversion to `'U'` dtype is unacceptable.

Examples

The examples below illustrate dealing with bytestring data in `astropy`.

```
>>> t = Table([['abc', 'def']], names=['a'], dtype=['S'])

>>> t['a'] == 'abc'  # Gives expected answer
array([ True, False], dtype=bool)

>>> t['a'] == b'abc'  # Still gives expected answer
array([ True, False], dtype=bool)

>>> t['a'][0] == 'abc'  # Expected answer
True

>>> t['a'][0] == b'abc'  # Cannot compare to bytestring
False

>>> t['a'][0] = 'bä'
>>> t
<Table length=2>
  a
bytes3
------
    bä
   def

>>> t['a'] == 'bä'
array([ True, False], dtype=bool)

>>> # Round trip unicode strings through HDF5
>>> t.write('test.hdf5', format='hdf5', path='data', overwrite=True)
>>> t2 = Table.read('test.hdf5', format='hdf5', path='data')
>>> t2
<Table length=2>
 col0
bytes3
------
    bä
   def
```

Modify Table

Modifying a Table

The data values within a **`Table`** object can be modified in much the same manner as for `numpy` structured arrays by accessing columns or rows of data and assigning values appropriately. A key enhancement provided by the **`Table`** class is the ability to modify the structure of the table: you can add or remove columns, and add new rows of data.

Quick Overview
The code below shows the basics of modifying a table and its data.

Examples
Make a table

```
>>> from astropy.table import Table
>>> import numpy as np
>>> arr = np.arange(15).reshape(5, 3)
>>> t = Table(arr, names=('a', 'b', 'c'), meta={'keywords': {'key1':
'val1'}})
```

Modify data values

```
>>> t['a'][:] = [1, -2, 3, -4, 5]  # Set all column values
>>> t['a'][2] = 30                 # Set row 2 of column 'a'
>>> t[1] = (8, 9, 10)              # Set all row values
>>> t[1]['b'] = -9                 # Set column 'b' of row 1
>>> t[0:3]['c'] = 100              # Set column 'c' of rows 0, 1, 2
```

Note that `table[row][column]` assignments will not work with `numpy` “fancy” `row` indexing (in that case `table[row]` would be a *copy* instead of a *view*). “Fancy” `numpy` indices include a **`list`**, **`numpy.ndarray`**, or **`tuple`** of **`numpy.ndarray`** (e.g., the return from **`numpy.where`**):

```
>>> t[[1, 2]]['a'] = [3., 5.]             # doesn't change table t
>>> t[np.array([1, 2])]['a'] = [3., 5.]   # doesn't change table t
>>> t[np.where(t['a'] > 3)]['a'] = 3.     # doesn't change table t
```

Instead use `table[column][row]` order:

```
>>> t['a'][[1, 2]] = [3., 5.]
>>> t['a'][np.array([1, 2])] = [3., 5.]
>>> t['a'][np.where(t['a'] > 3)] = 3.
```

You can also modify data columns with `unit` set in a way that follows the conventions of **`Quantity`** by using the **`quantity`** property:

```
>>> from astropy import units as u
>>> tu = Table([[1, 2.5]], names=('a',))
>>> tu['a'].unit = u.m
>>> tu['a'].quantity[:] = [1, 2] * u.km
>>> tu['a']
<Column name='a' dtype='float64' unit='m' length=2>
1000.0
2000.0
```

Add a column or columns

A single column can be added to a table using syntax like adding a dict value. The value on the right hand side can be a list or array of the correct size, or a scalar value that will be broadcast:

```
>>> t['d1'] = np.arange(5)
>>> t['d2'] = [1, 2, 3, 4, 5]
>>> t['d3'] = 6  # all 5 rows set to 6
```

For more explicit control, the **`add_column()`** and **`add_columns()`** methods can be used to add one or multiple columns to a table. In both cases the new column(s) can be specified as a list, an array (including **`Column`** or **`MaskedColumn`**), or a scalar:

```
>>> from astropy.table import Column
>>> t.add_column(np.arange(5), name='aa', index=0)  # Insert before
first table column
>>> t.add_column(1.0, name='bb')  # Add column of all 1.0 to end of
table
>>> c = Column(np.arange(5), name='e')
>>> t.add_column(c, index=0)  # Add Column using the existing column
name 'e'
>>> t.add_columns([[1, 2, 3, 4, 5], ['v', 'w', 'x', 'y', 'z']],
names=['h', 'i'])
```

Finally, columns can also be added from **`Quantity`** objects, which automatically sets the `.unit` attribute on the column:

```
>>> from astropy import units as u
>>> t['d'] = np.arange(1., 6.) * u.m
>>> t['d']
<Column name='d' dtype='float64' unit='m' length=5>
1.0
2.0
3.0
4.0
```

```
5.0
```

Remove columns

To remove a column from a table:

```
>>> t.remove_column('d1')
>>> t.remove_columns(['aa', 'd2', 'e'])
>>> del t['d3']
>>> del t['h', 'i']
>>> t.keep_columns(['a', 'b'])
```

Replace a column

You can entirely replace an existing column with a new column by setting the column to any object that could be used to initialize a table column (e.g., a list or `numpy` array). For example, you could change the data type of the `a` column from `int` to `float` using:

```
>>> t['a'] = t['a'].astype(float)
```

If the right-hand side value is not column-like, then an in-place update using broadcasting will be done, for example:

```
>>> t['a'] = 1  # Internally does t['a'][:] = 1
```

Rename columns

To rename a column:

```
>>> t.rename_column('a', 'a_new')
>>> t['b'].name = 'b_new'
```

Add a row of data

To add a row:

```
>>> t.add_row([-8, -9])
```

Remove rows

To remove a row:

```
>>> t.remove_row(0)
>>> t.remove_rows(slice(4, 5))
>>> t.remove_rows([1, 2])
```

Sort by one or more columns

To sort columns:

```
>>> t.sort('b_new')
>>> t.sort(['a_new', 'b_new'])
```

Reverse table rows

To reverse a table row:

```
>>> t.reverse()
```

Modify metadata

To modify metadata:

```
>>> t.meta['key'] = 'value'
```

Select or reorder columns

A new table with a subset or reordered list of columns can be created as shown in the following example:

```
>>> t = Table(arr, names=('a', 'b', 'c'))
>>> t_acb = t['a', 'c', 'b']
```

Another way to do the same thing is to provide a list or tuple as the item, as shown below:

```
>>> new_order = ['a', 'c', 'b']  # List or tuple
>>> t_acb = t[new_order]
```

Caveats

Modifying the table data and properties is fairly clear-cut, but one thing to keep in mind is that adding a row *may* require a new copy in memory of the table data. This depends on the detailed layout of Python objects in memory and cannot be reliably controlled. In some cases it may be possible to build a table row by row in less than O(N**2) time but you cannot count on it.

Another subtlety to keep in mind is that in some cases the return value of an operation results in a new table in memory while in other cases it results in a view of the existing table data. As an example, imagine trying to set two table elements using column selection with `t['a', 'c']` in combination with row index selection:

```
>>> t = Table([[1, 2], [3, 4], [5, 6]], names=('a', 'b', 'c'))
>>> t['a', 'c'][1] = (100, 100)
>>> print(t)
 a   b   c
--- --- ---
  1   3   5
  2   4   6
```

This might be surprising because the data values did not change and there was no error. In fact, what happened is that `t['a', 'c']` created a new temporary table in memory as a *copy* of the original and then updated the first row of the copy. The original `t` table was unaffected and the new temporary table disappeared once the statement was complete. The takeaway is to pay attention to how certain operations are performed one step at a time.

In-Place Versus Replace Column Update

Consider this code snippet:

```
>>> t = Table([[1, 2, 3]], names=['a'])
>>> t['a'] = [10.5, 20.5, 30.5]
```

There are a couple of ways this could be handled. It could update the existing array values in-place (truncating to integer), or it could replace the entire column with a new column based on the supplied data values.

The answer for `astropy` (since version 1.3) is that the operation shown above does a *complete replacement* of the column object. In this case it makes a new column object with float values by internally calling `t.replace_column('a', [10.5, 20.5, 30.5])`. In general this behavior is more consistent with Python and Pandas behavior.

Forcing in-place update

It is possible to force an in-place update of a column as follows:

```
t[colname][:] = value
```

Finding the source of problems

In order to find potential problems related to the replacing columns, there is a configuration option `table.conf.replace_warnings`. This controls a set of warnings that are emitted under certain circumstances when a table column is replaced. This option must be set to a list that includes zero or more of the following string values:

`always` :

Print a warning every time a column gets replaced via the setItem() syntax (i.e., `t['a'] = new_col`).

`slice` :
Print a warning when a column that appears to be a slice of a parent column is replaced.

`refcount` :
Print a warning when the Python reference count for the column changes. This indicates that a stale object exists that might be used elsewhere in the code and give unexpected results.

`attributes` :
Print a warning if any of the standard column attributes changed.

The default value for the `table.conf.replace_warnings` option is `[]` (no warnings).

Table Operations

Table Operations

In this section we describe high-level operations that can be used to generate a new table from one or more input tables. This includes:

Documentation	Description	Function
Grouped operations	Group tables and columns by keys	`group_by`
Binning	Binning tables	`group_by`
Stack vertically	Concatenate input tables along rows	`vstack`
Stack horizontally	Concatenate input tables along columns	`hstack`
Join	Database-style join of two tables	`join`
Unique rows	Unique table rows by keys	`unique`
Set difference	Set difference of two tables	`setdiff`
Table diff	Generic difference of two simple tables	`report_diff_values`

Grouped Operations

Sometimes in a table or table column there are natural groups within the dataset for which it makes sense to compute some derived values. A minimal example is a list of objects with photometry from various observing runs:

```
>>> from astropy.table import Table
>>> obs = Table.read("""name    obs_date    mag_b  mag_v
...                     M31     2012-01-02  17.0   17.5
...                     M31     2012-01-02  17.1   17.4
...                     M101    2012-01-02  15.1   13.5
...                     M82     2012-02-14  16.2   14.5
...                     M31     2012-02-14  16.9   17.3
...                     M82     2012-02-14  15.2   15.5
...                     M101    2012-02-14  15.0   13.6
...                     M82     2012-03-26  15.7   16.5
...                     M101    2012-03-26  15.1   13.5
...                     M101    2012-03-26  14.8   14.3
...                     """, format='ascii')
```

Table Groups

Now suppose we want the mean magnitudes for each object. We first group the data by the `name` column with the `group_by()` method. This returns a new table sorted by `name` which has a `groups` property specifying the unique values of `name` and the corresponding table rows:

```
>>> obs_by_name = obs.group_by('name')
>>> print(obs_by_name)
name  obs_date  mag_b mag_v
---- ---------- ----- -----
M101 2012-01-02  15.1  13.5  << First group (index=0, key='M101')
M101 2012-02-14  15.0  13.6
M101 2012-03-26  15.1  13.5
M101 2012-03-26  14.8  14.3
 M31 2012-01-02  17.0  17.5  << Second group (index=4, key='M31')
 M31 2012-01-02  17.1  17.4
 M31 2012-02-14  16.9  17.3
 M82 2012-02-14  16.2  14.5  << Third group (index=7, key='M83')
 M82 2012-02-14  15.2  15.5
 M82 2012-03-26  15.7  16.5
                             << End of groups (index=10)
>>> print(obs_by_name.groups.keys)
name
----
M101
 M31
 M82
>>> print(obs_by_name.groups.indices)
[ 0  4  7 10]
```

The `groups` property is the portal to all grouped operations with tables and columns. It defines how the table is grouped via an array of the unique row key values and the indices of the group boundaries for those key values. The groups here correspond to the row slices `0:4`, `4:7`, and `7:10` in the `obs_by_name` table.

The initial argument (`keys`) for the **group_by** function can take a number of input data types:

- Single string value with a table column name (as shown above)
- List of string values with table column names
- Another **Table** or **Column** with same length as table
- `numpy` structured array with same length as table
- `numpy` homogeneous array with same length as table

In all cases the corresponding row elements are considered as a tuple of values which form a key value that is used to sort the original table and generate the required groups.

As an example, to get the average magnitudes for each object on each observing night, we would first group the table on both `name` and `obs_date` as follows:

```
>>> print(obs.group_by(['name', 'obs_date']).groups.keys)
name  obs_date
---- ----------
M101 2012-01-02
M101 2012-02-14
M101 2012-03-26
 M31 2012-01-02
 M31 2012-02-14
 M82 2012-02-14
 M82 2012-03-26
```

Manipulating Groups

Once you have applied grouping to a table then you can access the individual groups or subsets of groups. In all cases this returns a new grouped table. For instance, to get the subtable which corresponds to the second group (index=1) do:

```
>>> print(obs_by_name.groups[1])
name  obs_date  mag_b mag_v
---- ---------- ----- -----
 M31 2012-01-02  17.0  17.5
 M31 2012-01-02  17.1  17.4
 M31 2012-02-14  16.9  17.3
```

To get the first and second groups together use a slice:

```
>>> groups01 = obs_by_name.groups[0:2]
>>> print(groups01)
name  obs_date  mag_b mag_v
---- ---------- ----- -----
M101 2012-01-02  15.1  13.5
M101 2012-02-14  15.0  13.6
M101 2012-03-26  15.1  13.5
M101 2012-03-26  14.8  14.3
 M31 2012-01-02  17.0  17.5
 M31 2012-01-02  17.1  17.4
 M31 2012-02-14  16.9  17.3
>>> print(groups01.groups.keys)
name
----
M101
 M31
```

You can also supply a `numpy` array of indices or a boolean mask to select particular groups, for example:

```
>>> mask = obs_by_name.groups.keys['name'] == 'M101'
>>> print(obs_by_name.groups[mask])
name  obs_date  mag_b mag_v
---- ---------- ----- -----
M101 2012-01-02  15.1  13.5
M101 2012-02-14  15.0  13.6
M101 2012-03-26  15.1  13.5
M101 2012-03-26  14.8  14.3
```

You can iterate over the group subtables and corresponding keys with:

```
>>> for key, group in zip(obs_by_name.groups.keys,
obs_by_name.groups):
...     print('****** {0} *******'.format(key['name']))
...     print(group)
...     print('')
...
****** M101 *******
name  obs_date  mag_b mag_v
---- ---------- ----- -----
M101 2012-01-02  15.1  13.5
M101 2012-02-14  15.0  13.6
M101 2012-03-26  15.1  13.5
M101 2012-03-26  14.8  14.3
****** M31 *******
name  obs_date  mag_b mag_v
---- ---------- ----- -----
```

```
 M31 2012-01-02  17.0  17.5
 M31 2012-01-02  17.1  17.4
 M31 2012-02-14  16.9  17.3
****** M82 *******
name  obs_date  mag_b mag_v
---- ---------- ----- -----
 M82 2012-02-14  16.2  14.5
 M82 2012-02-14  15.2  15.5
 M82 2012-03-26  15.7  16.5
```

Column Groups

Like `Table` objects, `Column` objects can also be grouped for subsequent manipulation with grouped operations. This can apply both to columns within a `Table` or bare `Column` objects.

As for `Table`, the grouping is generated with the `group_by` method. The difference here is that there is no option of providing one or more column names since that does not make sense for a `Column`.

Examples

To generate grouping in columns:

```
>>> from astropy.table import Column
>>> import numpy as np
>>> c = Column([1, 2, 3, 4, 5, 6], name='a')
>>> key_vals = np.array(['foo', 'bar', 'foo', 'foo', 'qux', 'qux'])
>>> cg = c.group_by(key_vals)

>>> for key, group in zip(cg.groups.keys, cg.groups):
...     print('****** {0} *******'.format(key))
...     print(group)
...     print('')
...
****** bar *******
 a
---
  2
****** foo *******
 a
---
  1
  3
  4
****** qux *******
 a
---
  5
  6
```

Aggregation

Aggregation is the process of applying a specified reduction function to the values within each group for each non-key column. This function must accept a `numpy` array as the first argument and return a single scalar value. Common function examples are **`numpy.sum`**, **`numpy.mean`**, and **`numpy.std`**.

For the example grouped table `obs_by_name` from above, we compute the group means with the **`aggregate`** method:

```
>>> obs_mean = obs_by_name.groups.aggregate(np.mean)
WARNING: Cannot aggregate column 'obs_date' [astropy.table.groups]
>>> print(obs_mean)
name mag_b mag_v
---- ----- ------
M101  15.0 13.725
 M31  17.0   17.4
 M82  15.7   15.5
```

It seems the magnitude values were successfully averaged, but what about the WARNING? Since the `obs_date` column is a string-type array, the **`numpy.mean`** function failed and raised an exception. Any time this happens then **`aggregate`** will issue a warning and then drop that column from the output result. Note that the `name` column is one of the `keys` used to determine the grouping so it is automatically ignored from aggregation.

From a grouped table it is possible to select one or more columns on which to perform the aggregation:

```
>>> print(obs_by_name['mag_b'].groups.aggregate(np.mean))
      mag_b
------------------
15.000000000000002
              17.0
15.699999999999998

>>> print(obs_by_name['name', 'mag_v',
'mag_b'].groups.aggregate(np.mean))
name       mag_v              mag_b
---- ------------------ ------------------
M101 13.725000000000001 15.000000000000002
 M31 17.400000000000002               17.0
 M82               15.5 15.699999999999998
```

A single column of data can be aggregated as well:

```
>>> c = Column([1, 2, 3, 4, 5, 6], name='a')
>>> key_vals = np.array(['foo', 'bar', 'foo', 'foo', 'qux', 'qux'])
>>> cg = c.group_by(key_vals)
>>> cg_sums = cg.groups.aggregate(np.sum)
>>> for key, cg_sum in zip(cg.groups.keys, cg_sums):
...     print('Sum for {0} = {1}'.format(key, cg_sum))
...
Sum for bar = 2
Sum for foo = 8
Sum for qux = 11
```

If the specified function has a `numpy.ufunc.reduceat` method, this will be called instead. This can improve the performance by a factor of 10 to 100 (or more) for large unmasked tables or columns with many relatively small groups. It also allows for the use of certain `` `numpy `` functions which normally take more than one input array but also work as reduction functions, like `numpy.add`. The `numpy` functions which should take advantage of using `numpy.ufunc.reduceat` include:

- `numpy.add`
- `numpy.arctan2`
- `numpy.bitwise_and`
- `numpy.bitwise_or`
- `numpy.bitwise_xor`
- `numpy.copysign`
- `numpy.divide`
- `numpy.equal`
- `numpy.floor_divide`
- `numpy.fmax`
- `numpy.fmin`
- `numpy.fmod`
- `numpy.greater_equal`
- `numpy.greater`
- `numpy.hypot`
- `numpy.left_shift`
- `numpy.less_equal`
- `numpy.less`
- `numpy.logaddexp2`
- `numpy.logaddexp`
- `numpy.logical_and`
- `numpy.logical_or`
- `numpy.logical_xor`
- `numpy.maximum`

- `numpy.minimum`
- `numpy.mod`
- `numpy.multiply`
- `numpy.not_equal`
- `numpy.power`
- `numpy.remainder`
- `numpy.right_shift`
- `numpy.subtract`
- `numpy.true_divide`

In special cases, `numpy.sum` and `numpy.mean` are substituted with their respective `reduceat` methods.

Filtering

Table groups can be filtered by means of the `filter` method. This is done by supplying a function which is called for each group. The function which is passed to this method must accept two arguments:

- `table` : `Table` object
- `key_colnames` : list of columns in `table` used as keys for grouping

It must then return either `True` or `False`.

Example

The following will select all table groups with only positive values in the non-key columns:

```
>>> def all_positive(table, key_colnames):
...     colnames = [name for name in table.colnames if name not in
key_colnames]
...     for colname in colnames:
...         if np.any(table[colname] < 0):
...             return False
...     return True
```

An example of using this function is:

```
>>> t = Table.read(""" a   b    c
...                   -2  7.0   0
...                   -2  5.0   1
...                    1  3.0  -5
...                    1 -2.0  -6
...                    1  1.0   7
...                    0  0.0   4
...                    3  3.0   5
...                    3 -2.0   6
...                       3  1.0   7""", format='ascii')
>>> tg = t.group_by('a')
>>> t_positive = tg.groups.filter(all_positive)
>>> for group in t_positive.groups:
...     print(group)
...     print('')
...
 a   b   c
--- --- ---
 -2 7.0   0
 -2 5.0   1

 a   b   c
--- --- ---
  0 0.0   4
```

As can be seen only the groups with `a == -2` and `a == 0` have all positive values in the non-key columns, so those are the ones that are selected.

Likewise a grouped column can be filtered with the **`filter`**, method but in this case the filtering function takes only a single argument which is the column group. It still must return either **`True`** or **`False`**. For example:

```
def all_positive(column):
    if np.any(column < 0):
        return False
    return True
```

Binning

A common tool in analysis is to bin a table based on some reference value. Examples:

- Photometry of a binary star in several bands taken over a span of time which should be binned by orbital phase.
- Reducing the sampling density for a table by combining 100 rows at a time.
- Unevenly sampled historical data which should binned to four points per year.

All of these examples of binning a table can be accomplished using grouped operations. The examples in that section are focused on the case of discrete key values such as the name of a source. In this section we show a concise yet powerful way of applying grouped operations to accomplish binning on key values such as time, phase, or row number.

The common theme in all of these cases is to convert the key value array into a new float- or int-valued array whose values are identical for rows in the same output bin.

Example

As an example, we generate a fake light curve:

```
>>> year = np.linspace(2000.0, 2010.0, 200)  # 200 observations over
10 years
>>> period = 1.811
>>> y0 = 2005.2
>>> mag = 14.0 + 1.2 * np.sin(2 * np.pi * (year - y0) / period)
>>> phase = ((year - y0) / period) % 1.0
>>> dat = Table([year, phase, mag], names=['year', 'phase', 'mag'])
```

Now we make an array that will be used for binning the data by 0.25 year intervals:

```
>>> year_bin = np.trunc(year / 0.25)
```

This has the property that all samples in each 0.25 year bin have, which is the same value of `year_bin`. Think of `year_bin` as the bin number for `year`. Then do the binning by grouping and immediately aggregating with `np.mean`.

```
>>> dat_grouped = dat.group_by(year_bin)
>>> dat_binned = dat_grouped.groups.aggregate(np.mean)
```

We can plot the results with `plt.plot(dat_binned['year'], dat_binned['mag'], '.')`. Alternately, we could bin into 10 phase bins:

```
>>> phase_bin = np.trunc(phase / 0.1)
>>> dat_grouped = dat.group_by(phase_bin)
>>> dat_binned = dat_grouped.groups.aggregate(np.mean)
```

This time, try plotting with `plt.plot(dat_binned['phase'], dat_binned['mag'])`.

Stack Vertically

The **`Table`** class supports stacking tables vertically with the **`vstack`** function. This process is also commonly known as concatenating or appending tables in the row direction. It corresponds roughly to the **`numpy.vstack`** function.

Examples

Suppose we have two tables of observations with several column names in common:

```
>>> from astropy.table import Table, vstack
>>> obs1 = Table.read("""name    obs_date    mag_b  logLx
...                      M31     2012-01-02  17.0   42.5
...                      M82     2012-10-29  16.2   43.5
...                      M101    2012-10-31  15.1   44.5""",
format='ascii')

>>> obs2 = Table.read("""name    obs_date    logLx
...                      NGC3516 2011-11-11  42.1
...                      M31     1999-01-05  43.1
...                      M82     2012-10-30  45.0""", format='ascii')
```

Now we can stack these two tables:

```
>>> print(vstack([obs1, obs2]))
  name   obs_date  mag_b logLx
------- ---------- ----- -----
    M31 2012-01-02  17.0  42.5
    M82 2012-10-29  16.2  43.5
   M101 2012-10-31  15.1  44.5
NGC3516 2011-11-11    --  42.1
    M31 1999-01-05    --  43.1
    M82 2012-10-30    --  45.0
```

Notice that the `obs2` table is missing the `mag_b` column, so in the stacked output table those values are marked as missing. This is the default behavior and corresponds to `join_type='outer'`. There are two other allowed values for the `join_type` argument, `'inner'` and `'exact'`:

```
>>> print(vstack([obs1, obs2], join_type='inner'))
  name   obs_date  logLx
------- ---------- -----
    M31 2012-01-02  42.5
    M82 2012-10-29  43.5
   M101 2012-10-31  44.5
NGC3516 2011-11-11  42.1
    M31 1999-01-05  43.1
    M82 2012-10-30  45.0

>>> print(vstack([obs1, obs2], join_type='exact'))
Traceback (most recent call last):
  ...
TableMergeError: Inconsistent columns in input arrays (use 'inner'
or 'outer' join_type to allow non-matching columns)
```

In the case of `join_type='inner'`, only the common columns (the intersection) are present in the output table. When `join_type='exact'` is specified, then **vstack** requires that all of the input tables have exactly the same column names.

More than two tables can be stacked by supplying a list of table objects:

```
>>> obs3 = Table.read("""name    obs_date    mag_b  logLx
...                     M45     2012-02-03  15.0   40.5""",
format='ascii')
>>> print(vstack([obs1, obs2, obs3]))
  name   obs_date  mag_b logLx
------- ---------- ----- -----
    M31 2012-01-02  17.0  42.5
    M82 2012-10-29  16.2  43.5
   M101 2012-10-31  15.1  44.5
NGC3516 2011-11-11    --  42.1
    M31 1999-01-05    --  43.1
    M82 2012-10-30    --  45.0
    M45 2012-02-03  15.0  40.5
```

See also the sections on Merging metadata and Merging column attributes for details on how these characteristics of the input tables are merged in the single output table. Note also that you can use a single table row instead of a full table as one of the inputs.

Stack Horizontally

The **`Table`** class supports stacking tables horizontally (in the column-wise direction) with the **`hstack`** function. It corresponds roughly to the **`numpy.hstack`** function.

Examples

Suppose we have the following two tables:

```
>>> from astropy.table import Table, hstack
>>> t1 = Table.read("""a   b    c
...                    1   foo  1.4
...                    2   bar  2.1
...                    3   baz  2.8""", format='ascii')
>>> t2 = Table.read("""d     e
...                    ham   eggs
...                    spam  toast""", format='ascii')
```

Now we can stack these two tables horizontally:

```
>>> print(hstack([t1, t2]))
 a   b   c   d     e
--- --- --- ---- -----
  1 foo 1.4  ham  eggs
  2 bar 2.1 spam toast
  3 baz 2.8   --    --
```

As with **vstack**, there is an optional `join_type` argument that can take values `'inner'`, `'exact'`, and `'outer'`. The default is `'outer'`, which effectively takes the union of available rows and masks out any missing values. This is illustrated in the example above. The other options give the intersection of rows, where `'exact'` requires that all tables have exactly the same number of rows:

```
>>> print(hstack([t1, t2], join_type='inner'))
 a   b   c   d     e
--- --- --- ---- -----
  1 foo 1.4  ham  eggs
  2 bar 2.1 spam toast

>>> print(hstack([t1, t2], join_type='exact'))
Traceback (most recent call last):
  ...
TableMergeError: Inconsistent number of rows in input arrays (use
'inner' or
'outer' join_type to allow non-matching rows)
```

More than two tables can be stacked by supplying a list of table objects. The example below also illustrates the behavior when there is a conflict in the input column names (see the section on Column renaming for details):

```
>>> t3 = Table.read("""a    b
...                    M45  2012-02-03""", format='ascii')
>>> print(hstack([t1, t2, t3]))
a_1 b_1  c   d     e   a_3    b_3
--- --- --- ---- ----- --- ----------
  1 foo 1.4  ham  eggs M45 2012-02-03
  2 bar 2.1 spam toast  --         --
  3 baz 2.8   --    --  --         --
```

The metadata from the input tables is merged by the process described in the Merging metadata section. Note also that you can use a single table row instead of a full table as one of the inputs.

Stack Depth-Wise

The **Table** class supports stacking columns within tables depth-wise using the **dstack** function. It corresponds roughly to running the **numpy.dstack** function on the individual columns matched by name.

Examples

Suppose we have tables of data for sources giving information on the enclosed source counts for different PSF fractions:

```
>>> from astropy.table import Table, dstack
>>> src1 = Table.read("""psf_frac  counts
...                      0.10        45
...                      0.50        90
...                      0.90       120
...                      """, format='ascii')

>>> src2 = Table.read("""psf_frac  counts
...                      0.10       200
...                      0.50       300
...                      0.90       350
...                      """, format='ascii')
```

Now we can stack these two tables depth-wise to get a single table with the characteristics of both sources:

```
>>> srcs = dstack([src1, src2])
>>> print(srcs)
psf_frac [2] counts [2]
------------ ----------
  0.1 .. 0.1  45 .. 200
  0.5 .. 0.5  90 .. 300
  0.9 .. 0.9 120 .. 350
```

In this case the counts for the first source are accessible as `srcs['counts'][:, 0]`, and likewise the second source counts are `srcs['counts'][:, 1]`.

For this function the length of all input tables must be the same. This function can accept `join_type` and `metadata_conflicts` just like the **`vstack`** function. The `join_type` argument controls how to handle mismatches in the columns of the input table.

See also the sections on Merging metadata and Merging column attributes for details on how these characteristics of the input tables are merged in the single output table. Note also that you can use a single table row instead of a full table as one of the inputs.

Join

The **`Table`** class supports the database join operation. This provides a flexible and powerful way to combine tables based on the values in one or more key columns.

Examples

Suppose we have two tables of observations, the first with B and V magnitudes and the second with X-ray luminosities of an overlapping (but not identical) sample:

```
>>> from astropy.table import Table, join
>>> optical = Table.read("""name    obs_date    mag_b  mag_v
...                        M31     2012-01-02  17.0   16.0
...                        M82     2012-10-29  16.2   15.2
...                        M101    2012-10-31  15.1   15.5""",
format='ascii')
>>> xray = Table.read("""   name    obs_date    logLx
...                        NGC3516 2011-11-11  42.1
...                        M31     1999-01-05  43.1
...                        M82     2012-10-29  45.0""",
format='ascii')
```

The `join()` method allows you to merge these two tables into a single table based on matching values in the "key columns". By default, the key columns are the set of columns that are common to both tables. In this case the key columns are `name` and `obs_date`. We can find all of the observations of the same object on the same date as follows:

```
>>> opt_xray = join(optical, xray)
>>> print(opt_xray)
name  obs_date  mag_b mag_v logLx
---- ---------- ----- ----- -----
 M82 2012-10-29  16.2  15.2  45.0
```

We can perform the match by `name` only by providing the `keys` argument, which can be either a single column name or a list of column names:

```
>>> print(join(optical, xray, keys='name'))
name obs_date_1 mag_b mag_v obs_date_2 logLx
---- ---------- ----- ----- ---------- -----
 M31 2012-01-02  17.0  16.0 1999-01-05  43.1
 M82 2012-10-29  16.2  15.2 2012-10-29  45.0
```

This output table has all of the observations that have both optical and X-ray data for an object (M31 and M82). Notice that since the `obs_date` column occurs in both tables, it has been split into two columns, `obs_date_1` and `obs_date_2`. The values are taken from the "left" (`optical`) and "right" (`xray`) tables, respectively.

Different Join Options

The table joins so far are known as "inner" joins and represent the strict intersection of the two tables on the key columns.

If you want to make a new table which has *every* row from the left table and includes matching values from the right table when available, this is known as a left join:

```
>>> print(join(optical, xray, join_type='left'))
name  obs_date  mag_b mag_v logLx
---- ---------- ----- ----- -----
M101 2012-10-31  15.1  15.5    --
 M31 2012-01-02  17.0  16.0    --
 M82 2012-10-29  16.2  15.2  45.0
```

Two of the observations do not have X-ray data, as indicated by the `--` in the table. When there are any missing values the output will be a masked table (see Masking and Missing Values for more information). You might be surprised that there is no X-ray data for M31 in the output. Remember that the default matching key includes both `name` and `obs_date`. Specifying the key as only the `name` column gives:

```
>>> print(join(optical, xray, join_type='left', keys='name'))
name obs_date_1 mag_b mag_v obs_date_2 logLx
---- ---------- ----- ----- ---------- -----
M101 2012-10-31  15.1  15.5         --    --
 M31 2012-01-02  17.0  16.0 1999-01-05  43.1
 M82 2012-10-29  16.2  15.2 2012-10-29  45.0
```

Likewise you can construct a new table with every row of the right table and matching left values (when available) using `join_type='right'`.

To make a table with the union of rows from both tables do an “outer” join:

```
>>> print(join(optical, xray, join_type='outer'))
  name   obs_date  mag_b mag_v logLx
------- ---------- ----- ----- -----
   M101 2012-10-31  15.1  15.5    --
    M31 1999-01-05    --    --  43.1
    M31 2012-01-02  17.0  16.0    --
    M82 2012-10-29  16.2  15.2  45.0
NGC3516 2011-11-11    --    --  42.1
```

In all the above cases the output join table will be sorted by the key column(s) and in general will not preserve the row order of the input tables.

Finally, you can do a “Cartesian” join, which is the Cartesian product of all available rows. In this case one there are no key columns (and supplying a `keys` argument is an error):

```
>>> print(join(optical, xray, join_type='cartesian'))
name_1 obs_date_1 mag_b mag_v  name_2 obs_date_2 logLx
------ ---------- ----- ----- ------- ---------- -----
   M31 2012-01-02  17.0  16.0 NGC3516 2011-11-11  42.1
   M31 2012-01-02  17.0  16.0     M31 1999-01-05  43.1
   M31 2012-01-02  17.0  16.0     M82 2012-10-29  45.0
   M82 2012-10-29  16.2  15.2 NGC3516 2011-11-11  42.1
   M82 2012-10-29  16.2  15.2     M31 1999-01-05  43.1
   M82 2012-10-29  16.2  15.2     M82 2012-10-29  45.0
  M101 2012-10-31  15.1  15.5 NGC3516 2011-11-11  42.1
  M101 2012-10-31  15.1  15.5     M31 1999-01-05  43.1
  M101 2012-10-31  15.1  15.5     M82 2012-10-29  45.0
```

Non-Identical Key Column Names

The `join()` function requires the key column names to be identical in the two tables. However, in the following, one table has a `'name'` column while the other has an `'obj_id'` column:

```
>>> optical = Table.read("""name    obs_date    mag_b  mag_v
...                         M31     2012-01-02  17.0   16.0
...                         M82     2012-10-29  16.2   15.2
...                         M101    2012-10-31  15.1   15.5""",
format='ascii')
>>> xray_1 = Table.read("""   obj_id    obs_date    logLx
...                           NGC3516 2011-11-11  42.1
...                           M31     1999-01-05  43.1
...                           M82     2012-10-29  45.0""",
format='ascii')
```

In order to perform a match based on the names of the objects, you have to temporarily rename one of the columns mentioned above, right before creating the new table:

```
>>> xray_1.rename_column('obj_id', 'name')
>>> opt_xray_1 = join(optical, xray_1, keys='name')
>>> xray_1.rename_column('name', 'obj_id')
>>> print(opt_xray_1)
name obs_date_1 mag_b mag_v obs_date_2 logLx
---- ---------- ----- ----- ---------- -----
M31 2012-01-02  17.0  16.0 1999-01-05  43.1
M82 2012-10-29  16.2  15.2 2012-10-29  45.0
```

The original `xray_1` table remains unchanged after the operation:

```
>>> print(xray_1)
obj_id  obs_date  logLx
```

```
------- ---------- -----
NGC3516 2011-11-11  42.1
    M31 1999-01-05  43.1
    M82 2012-10-29  45.0
```

Identical Key Values

The **Table** join operation works even if there are multiple rows with identical key values. For example, the following tables have multiple rows for the key column `x`:

```
>>> from astropy.table import Table, join
>>> left = Table([[0, 1, 1, 2], ['L1', 'L2', 'L3', 'L4']], names=
('key', 'L'))
>>> right = Table([[1, 1, 2, 4], ['R1', 'R2', 'R3', 'R4']], names=
('key', 'R'))
>>> print(left)
key  L
--- ---
  0  L1
  1  L2
  1  L3
  2  L4
>>> print(right)
key  R
--- ---
  1  R1
  1  R2
  2  R3
  4  R4
```

Doing an outer join on these tables shows that what is really happening is a Cartesian product. For each matching key, every combination of the left and right tables is represented. When there is no match in either the left or right table, the corresponding column values are designated as missing.

```
>>> print(join(left, right, join_type='outer'))
key  L   R
--- --- ---
  0  L1  --
  1  L2  R1
  1  L2  R2
  1  L3  R1
  1  L3  R2
  2  L4  R3
  4  --  R4
```

Note

The output table is sorted on the key columns, but when there are rows with identical keys the output order in the non-key columns is not guaranteed to be identical across installations. In the example above, the order within the four rows with `key == 1` can vary.

An inner join is the same but only returns rows where there is a key match in both the left and right tables:

```
>>> print(join(left, right, join_type='inner'))
key  L   R
--- --- ---
  1  L2  R1
  1  L2  R2
  1  L3  R1
  1  L3  R2
  2  L4  R3
```

Conflicts in the input table names are handled by the process described in the section on Column renaming. See also the sections on Merging metadata and Merging column attributes for details on how these characteristics of the input tables are merged in the single output table.

Merging Details

When combining two or more tables there is the need to merge certain characteristics in the inputs and potentially resolve conflicts. This section describes the process.

Column Renaming

In cases where the input tables have conflicting column names, there is a mechanism to generate unique output column names. There are two keyword arguments that control the renaming behavior:

`table_names`
: Two-element list of strings that provide a name for the tables being joined. By default this is `['1', '2', ...]`, where the numbers correspond to the input tables.

`uniq_col_name`
: String format specifier with a default value of `'{col_name}_{table_name}'`.

This is best understood by example using the `optical` and `xray` tables in the **`join()`** example defined previously:

```
>>> print(join(optical, xray, keys='name',
...            table_names=['OPTICAL', 'XRAY'],
...            uniq_col_name='{table_name}_{col_name}'))
name OPTICAL_obs_date mag_b mag_v XRAY_obs_date logLx
---- ---------------- ----- ----- ------------- -----
 M31       2012-01-02  17.0  16.0    1999-01-05  43.1
 M82       2012-10-29  16.2  15.2    2012-10-29  45.0
```

Merging Metadata

`Table` objects can have associated metadata:

- `Table.meta` : table-level metadata as an ordered dictionary
- `Column.meta` : per-column metadata as an ordered dictionary

The table operations described here handle the task of merging the metadata in the input tables into a single output structure. Because the metadata can be arbitrarily complex there is no unique way to do the merge. The current implementation uses a recursive algorithm with four rules:

- `dict` elements are merged by keys.
- Conflicting `list` or `tuple` elements are concatenated.
- Conflicting `dict` elements are merged by recursively calling the merge function.
- Conflicting elements that are not both `list`, `tuple`, or `dict` will follow the following rules:
    - If both metadata values are identical, the output is set to this value.
    - If one of the conflicting metadata values is `None`, the other value is picked.
    - If both metadata values are different and neither is `None`, the one for the last table in the list is picked.

By default, a warning is emitted in the last case (both metadata values are not `None`). The warning can be silenced or made into an exception using the `metadata_conflicts` argument to `hstack()`, `vstack()`, or `join()`. The `metadata_conflicts` option can be set to:

- `'silent'` – no warning is emitted, the value for the last table is silently picked.
- `'warn'` – a warning is emitted, the value for the last table is picked.
- `'error'` – an exception is raised.

The default strategies for merging metadata can be augmented or customized by defining subclasses of the `MergeStrategy` base class. In most cases you will also use the `enable_merge_strategies` for enabling the custom strategies. The linked documentation strings provide details.

Merging Column Attributes

In addition to the table and column `meta` attributes, the column attributes `unit`, `format`, and `description` are merged by going through the input tables in order and taking the first value which is defined (i.e., is not None).

Example

To merge column attributes `unit`, `format`, or `description`:

```
>>> from astropy.table import Column, Table, vstack
>>> col1 = Column([1], name='a')
>>> col2 = Column([2], name='a', unit='cm')
>>> col3 = Column([3], name='a', unit='m')
>>> t1 = Table([col1])
>>> t2 = Table([col2])
>>> t3 = Table([col3])
>>> out = vstack([t1, t2, t3])
WARNING: MergeConflictWarning: In merged column 'a' the 'unit'
attribute does
not match (cm != m).  Using m for merged output
[astropy.table.operations]
>>> out['a'].unit
Unit("m")
```

The rules for merging are the same as for Merging metadata, and the `metadata_conflicts` option also controls the merging of column attributes.

Joining Coordinates and Custom Join Functions

If you have two source catalogs that have `SkyCoord` coordinate columns, these can be joined using cross-matching of the coordinates with a specified distance threshold. This is a special case of a more general problem of “fuzzy” matching of key column values, where instead of an exact match we require only an approximate match. This is supported using the `join_funcs` argument (introduced in version 4.1).

Example

To join two tables on a `SkyCoord` key column we use the `join_funcs` keyword to supply a `dict` of functions that specify how to match a particular key column by name. In the example below we are joining on the `sc` column, so we provide the following argument:

```
join_funcs={'sc': join_skycoord(0.2 * u.deg)}
```

This tells `join` to match the `sc` key column using a custom join function `join_skycoord` using a matching distance threshold of 0.2 deg. Under the hood this calls `search_around_sky` or `search_around_3d` to do the cross-matching. The default is using `'search_around_sky'` (angle) matching, but `'search_around_3d'` (length or dimensionless) is also available. This is specified using the `distance_func` argument of `join_skycoord`, which can also be a function that matches the input and output API of `search_around_sky`.

Now we show the whole process:

```
>>> from astropy.coordinates import SkyCoord
>>> import astropy.units as u
>>> from astropy.table import Table, join, join_skycoord
```

```
>>> sc1 = SkyCoord([0, 1, 1.1, 2], [0, 0, 0, 0], unit='deg')
>>> sc2 = SkyCoord([1.05, 0.5, 2.1], [0, 0, 0], unit='deg')
```

```
>>> t1 = Table([sc1, [0, 1, 2, 3]], names=['sc', 'idx_1'])
>>> t2 = Table([sc2, [0, 1, 2]], names=['sc', 'idx_2'])
```

```
>>> t12 = join(t1, t2, join_funcs={'sc': join_skycoord(0.2 * u.deg)})
>>> print(t12)
sc_id   sc_1  idx_1   sc_2   idx_2
      deg,deg        deg,deg
----- ------- ----- -------- -----
    1 1.0,0.0     1 1.05,0.0     0
    1 1.1,0.0     2 1.05,0.0     0
    2 2.0,0.0     3  2.1,0.0     2
```

The joined table has matched the sources within 0.2 deg and created a new column `sc_id` with a unique identifier for each source.

You might be wondering what is happening in the join function defined above, especially if you are interested in defining your own such function. This could be done in order to allow fuzzy word matching of tables, for example joining tables of people by name where the names do not always match exactly.

The first thing to note here is that the `join_skycoord` function actually returns a function itself. This allows specifying a variable match distance via a function enclosure. The requirement of the join function is that it accepts two arguments corresponding to the two key columns, and returns a tuple of `(ids1, ids2)`. These identifiers correspond to the identification of each column entry with a unique matched source.

```
>>> join_func = join_skycoord(0.2 * u.deg)
>>> join_func(sc1, sc2)  # Associate each coordinate with unique
source ID
(array([3, 1, 1, 2]), array([1, 4, 2]))
```

If you would like to write your own fuzzy matching function, we suggest starting from the source code for `join_skycoord` or `join_distance`.

Join on Distance

The example above focused on joining on a `SkyCoord`, but you can also join on a generic distance between column values using the `join_distance` join function. This can apply to 1D or 2D (vector) columns. This will look very similar to the coordinates example, but here there is a bit more flexibility. The matching is done using `scipy.spatial.cKDTree` and `scipy.spatial.cKDTree.query_ball_tree`, and the behavior of these can be controlled via the `kdtree_args` and `query_args` arguments, respectively.

Unique Rows

Sometimes it makes sense to use only rows with unique key columns or even fully unique rows from a table. This can be done using the above described `group_by()` method and `groups` attribute, or with the `unique` convenience function. The `unique` function returns with a sorted table containing the first row for each unique `keys` column value. If no `keys` is provided, it returns with a sorted table containing all of the fully unique rows.

Example

An example of a situation where you might want to use rows with unique key columns is a list of objects with photometry from various observing runs. Using `'name'` as the only `keys`, it returns with the first occurrence of each of the three targets:

```
>>> from astropy import table
>>> obs = table.Table.read("""name    obs_date    mag_b  mag_v
...                          M31     2012-01-02  17.0   17.5
...                          M82     2012-02-14  16.2   14.5
...                          M101    2012-01-02  15.1   13.5
...                          M31     2012-01-02  17.1   17.4
...                          M101    2012-01-02  15.1   13.5
...                          M82     2012-02-14  16.2   14.5
...                          M31     2012-02-14  16.9   17.3
...                          M82     2012-02-14  15.2   15.5
...                          M101    2012-02-14  15.0   13.6
...                          M82     2012-03-26  15.7   16.5
...                          M101    2012-03-26  15.1   13.5
...                           M101    2012-03-26  14.8   14.3
...                           """, format='ascii')
>>> unique_by_name = table.unique(obs, keys='name')
>>> print(unique_by_name)
name  obs_date  mag_b mag_v
---- ---------- ----- -----
M101 2012-01-02  15.1  13.5
 M31 2012-01-02  17.0  17.5
 M82 2012-02-14  16.2  14.5
```

Using multiple columns as `keys`:

```
>>> unique_by_name_date = table.unique(obs, keys=['name',
'obs_date'])
>>> print(unique_by_name_date)
name  obs_date  mag_b mag_v
---- ---------- ----- -----
M101 2012-01-02  15.1  13.5
M101 2012-02-14  15.0  13.6
M101 2012-03-26  15.1  13.5
 M31 2012-01-02  17.0  17.5
 M31 2012-02-14  16.9  17.3
 M82 2012-02-14  16.2  14.5
 M82 2012-03-26  15.7  16.5
```

Set Difference

A set difference will tell you the elements that are contained in one set but not in the other. This concept can be applied to rows of a table by using the **`setdiff`** function. You provide the function with two input tables and it will return all rows in the first table which do not occur in the second table.

The optional `keys` parameter specifies the names of columns that are used to match table rows. This can be a subset of the full list of columns, but both the first and second tables must contain all columns specified by `keys`. If not provided, then `keys` defaults to all column names in the first table.

If no different rows are found, the **`setdiff`** function will return an empty table.

Example

The example below illustrates finding the set difference of two observation lists using a common subset of the columns in two tables.:

```
>>> from astropy.table import Table, setdiff
>>> cat_1 = Table.read("""name    obs_date    mag_b   mag_v
...                       M31     2012-01-02  17.0    16.0
...                       M82     2012-10-29  16.2    15.2
...                       M101    2012-10-31  15.1    15.5""",
format='ascii')
>>> cat_2 = Table.read("""   name    obs_date    logLx
...                            NGC3516 2011-11-11  42.1
...                            M31     2012-01-02  43.1
...                            M82     2012-10-29  45.0""",
format='ascii')
>>> sdiff = setdiff(cat_1, cat_2, keys=['name', 'obs_date'])
>>> print(sdiff)
name  obs_date  mag_b mag_v
---- ---------- ----- -----
M101 2012-10-31  15.1  15.5
```

In this example there is a column in the first table that is not present in the second table, so the `keys` parameter must be used to specify the desired column names.

Table Diff

To compare two tables, you can use **`report_diff_values()`**, which would produce a report identical to FITS diff.

Example

The example below illustrates finding the difference between two tables:

```
>>> from astropy.table import Table
>>> from astropy.utils.diff import report_diff_values
>>> import sys
>>> cat_1 = Table.read("""name    obs_date    mag_b  mag_v
...                       M31     2012-01-02  17.0   16.0
...                       M82     2012-10-29  16.2   15.2
...                       M101    2012-10-31  15.1   15.5""",
format='ascii')
>>> cat_2 = Table.read("""name    obs_date    mag_b  mag_v
...                       M31     2012-01-02  17.0   16.5
...                       M82     2012-10-29  16.2   15.2
...                       M101    2012-10-30  15.1   15.5
...                       NEW     2018-05-08   nan    9.0""",
format='ascii')
>>> identical = report_diff_values(cat_1, cat_2, fileobj=sys.stdout)
     name  obs_date  mag_b mag_v
     ---- ---------- ----- -----
  a>  M31 2012-01-02  17.0  16.0
   ?                           ^
  b>  M31 2012-01-02  17.0  16.5
   ?                           ^
      M82 2012-10-29  16.2  15.2
  a> M101 2012-10-31  15.1  15.5
   ?                ^
  b> M101 2012-10-30  15.1  15.5
```

```
   ?                ^
  b>  NEW 2018-05-08   nan   9.0
>>> identical
False
```

Indexing

Table Indexing

Once a **Table** has been created, it is possible to create indexes on one or more columns of the table. An index internally sorts the rows of a table based on the index column(s), allowing for element retrieval by column value and improved performance for certain table operations.

Creating an Index

To create an index on a table, use the **add_index()** method:

```
>>> from astropy.table import Table
>>> t = Table([(2, 3, 2, 1), (8, 7, 6, 5)], names=('a', 'b'))
>>> t.add_index('a')
```

The optional argument `unique` may be specified to create an index with uniquely valued elements.

To create a composite index on multiple columns, pass a list of columns instead:

```
>>> t.add_index(['a', 'b'])
```

In particular, the first index created using the **add_index()** method is considered the default index or the “primary key.” To retrieve an index from a table, use the **indices** property:

```
>>> t.indices['a']
 a   rows
--- ----
  1    3
  2    0
  2    2
  3    1
>>> t.indices['a', 'b']
 a   b   rows
--- --- ----
```

```
  1   5    3
  2   6    2
  2   8    0
  3   7    1
```

Row Retrieval using Indices

Row retrieval can be accomplished using two table properties: **loc** and **iloc**. The **loc** property can be indexed either by column value, range of column values (*including* the bounds), or a list or ndarray of column values:

```
>>> t = Table([(1, 2, 3, 4), (10, 1, 9, 9)], names=('a', 'b'), dtype=
['i8', 'i8'])
>>> t.add_index('a')
>>> t.loc[2]
<Row index=1>
  a     b
int64 int64
----- -----
    2     1
>>> t.loc[[1, 4]]
<Table length=2>
  a     b
int64 int64
----- -----
    1    10
    4     9
>>> t.loc[1:3]
<Table length=3>
  a     b
int64 int64
----- -----
    1    10
    2     1
    3     9
>>> t.loc[:]
<Table length=4>
  a     b
int64 int64
----- -----
    1    10
    2     1
    3     9
    4     9
```

Note that by default, **loc** uses the primary index, which here is column ‘a’. To use a different index, pass the indexed column name before the retrieval data:

```
>>> t.add_index('b')
>>> t.loc['b', 8:10]
<Table length=3>
  a     b
int64 int64
----- -----
    3     9
    4     9
    1    10
```

The property **iloc** works similarly, except that the retrieval information must be either an integer or a slice, and relates to the sorted order of the index rather than column values. For example:

```
>>> t.iloc[0] # smallest row by value 'a'
<Row index=0>
  a     b
int64 int64
----- -----
    1    10
>>> t.iloc['b', 1:] # all but smallest value of 'b'
<Table length=3>
  a     b
int64 int64
----- -----
    3     9
    4     9
    1    10
```

Effects on Performance

Table operations change somewhat when indices are present, and there are a number of factors to consider when deciding whether the use of indices will improve performance. In general, indexing offers the following advantages:

- Table grouping and sorting based on indexed column(s) both become faster.
- Retrieving values by index is faster than custom searching.

There are certain caveats, however:

- Creating an index requires time and memory.
- Table modifications become slower due to automatic index updates.
- Slicing a table becomes slower due to index relabeling.

See here for an IPython notebook profiling various aspects of table indexing.

Index Modes

The **index_mode()** method allows for some flexibility in the behavior of table indexing by allowing the user to enter a specific indexing mode via a context manager. There are currently three indexing modes: `freeze`, `copy_on_getitem`, and `discard_on_copy`.

The `freeze` mode prevents automatic index updates whenever a column of the index is modified, and all indices refresh themselves after the context ends:

```
>>> with t.index_mode('freeze'):
...    t['a'][0] = 0
...    print(t.indices['a']) # unmodified
 a  rows
--- ----
  1    0
  2    1
  3    2
  4    3
>>> print(t.indices['a']) # modified
 a  rows
--- ----
  0    0
  2    1
  3    2
  4    3
```

The `copy_on_getitem` mode forces columns to copy and relabel their indices upon slicing. In the absence of this mode, table slices will preserve indices while column slices will not:

```
>>> ca = t['a'][[1, 3]]
>>> ca.info.indices
[]
>>> with t.index_mode('copy_on_getitem'):
...     ca = t['a'][[1, 3]]
...     print(ca.info.indices)
[ a  rows
--- ----
  2    0
  4    1]
```

The `discard_on_copy` mode prevents indices from being copied whenever a column or table is copied:

```
>>> t2 = Table(t)
>>> t2.indices['a']
 a  rows
--- ----
  0    0
  2    1
  3    2
  4    3
>>> t2.indices['b']
 b  rows
--- ----
  1    1
  9    2
  9    3
 10    0
>>> with t.index_mode('discard_on_copy'):
...     t2 = Table(t)
...     print(t2.indices)
[]
```

Updating Rows using Indices

Row updates can be accomplished by assigning the table property `loc` a complete row or a list of rows:

```
>>> t = Table([('w', 'x', 'y', 'z'), (10, 1, 9, 9)], names=('a',
'b'), dtype=['str', 'i8'])
>>> t.add_index('a')
>>> t.loc['x']
<Row index=1>
 a     b
str1 int64
---- -----
   x     1
>>> t.loc['x'] = ['a', 12]
>>> t
<Table length=4>
 a     b
str1 int64
---- -----
   w    10
   a    12
   y     9
   z     9
>>> t.loc[['w', 'y']]
<Table length=2>
 a     b
str1 int64
---- -----
   w    10
   y     9
>>> t.loc[['w', 'z']] = [['b',23], ['c',56]]
>>> t
<Table length=4>
```

```
 a     b
str1 int64
---- -----
   b    23
   a    12
   y     9
   c    56
```

Retrieving the Location of Rows using Indices

Retrieval of the location of rows can be accomplished using a table property: **loc_indices**. The **loc_indices** property can be indexed either by column value, range of column values (*including* the bounds), or a list or ndarray of column values:

```
>>> t = Table([('w', 'x', 'y', 'z'), (10, 1, 9, 9)], names=('a',
'b'), dtype=['str', 'i8'])
>>> t.add_index('a')
>>> t.loc_indices['x']
1
```

Engines

When creating an index via **add_index()**, the keyword argument `engine` may be specified to use a particular indexing engine. The available engines are:

- **SortedArray**, a sorted array engine using an underlying sorted `Table`.
- **SCEngine**, a sorted list engine using the Sorted Containers package.
- **BST**, a Python-based binary search tree engine.

The SCEngine depends on the `sortedcontainers` dependency. The most important takeaway is that **SortedArray** (the default engine) is usually best, although **SCEngine** may be more appropriate for an index created on an empty column since adding new values is quicker.

Masking

Masking and Missing Values

The **astropy.table** package provides support for masking and missing values in a table by using the `numpy.ma` masked array package to define masked columns and by supporting Mixin Columns that provide masking. This allows handling tables with missing or invalid entries in much the same manner as for standard (unmasked) tables. It is useful to be familiar with the masked array documentation when using masked tables within `astropy.table`.

In a nutshell, the concept is to define a boolean mask that mirrors the structure of a column data array. Wherever a mask value is `True`, the corresponding entry is considered to be missing or invalid. Operations involving column or row access and slicing are unchanged. The key difference is that arithmetic or reduction operations involving columns or column slices follow the rules for operations on masked arrays.

Important

Changes in `astropy` 4.0

In `astropy` 4.0 the behavior of masked tables was changed in a way that could impact program functionality. See Masking Change in astropy 4.0 for details.

Note

Reduction operations like `numpy.sum` or `numpy.mean` follow the convention of ignoring masked (invalid) values. This differs from the behavior of the floating point `NaN`, for which the sum of an array including one or more `NaN's` will result in `NaN`.

See this page for information on NumPy Enhancement Proposals 24, 25, and 26.

Table Creation

A masked table can be created in several ways:

Create a table with one or more columns as a MaskedColumn object

```
>>> from astropy.table import Table, Column, MaskedColumn
>>> a = MaskedColumn([1, 2], name='a', mask=[False, True],
dtype='i4')
>>> b = Column([3, 4], name='b', dtype='i8')
>>> Table([a, b])
<Table length=2>
  a     b
int32 int64
----- -----
    1     3
   --     4
```

The `MaskedColumn` is the masked analog of the `Column` class and provides the interface for creating and manipulating a column of masked data. The `MaskedColumn` class inherits from `numpy.ma.MaskedArray`, in contrast to

Column which inherits from numpy.ndarray. This distinction is the main reason there are different classes for these two cases.

Notice that masked entries in the table output are shown as `--`.

Create a table with one or more columns as a ``numpy`` MaskedArray

```
>>> import numpy as np
>>> a = np.ma.array([1, 2])
>>> b = [3, 4]
>>> t = Table([a, b], names=('a', 'b'))
```

Create a table from list data containing `numpy.ma.masked`

You can use the numpy.ma.masked constant to indicate masked or invalid data:

```
>>> a = [1.0, np.ma.masked]
>>> b = [np.ma.masked, 'val']
>>> Table([a, b], names=('a', 'b'))
<Table length=2>
  a      b
float64 str3
------- ----
    1.0   --
    --   val
```

Initializing from lists with embedded numpy.ma.masked elements is considerably slower than using numpy.ma.array or MaskedColumn directly, so if performance is a concern you should use the latter methods if possible.

Add a MaskedColumn object to an existing table

```
>>> t = Table([[1, 2]], names=['a'])
>>> b = MaskedColumn([3, 4], mask=[True, False])
>>> t['b'] = b
```

Add a new row to an existing table and specify a mask argument

```
>>> a = Column([1, 2], name='a')
>>> b = Column([3, 4], name='b')
>>> t = Table([a, b])
>>> t.add_row([3, 6], mask=[True, False])
```

Create a new table object and specify masked=True

If `masked=True` is provided when creating the table then every column will be created as a MaskedColumn, and new columns will always be added as a a

MaskedColumn.

```
>>> Table([(1, 2), (3, 4)], names=('a', 'b'), masked=True, dtype=
('i4', 'i8'))
<Table masked=True length=2>
  a     b
int32 int64
----- -----
    1     3
    2     4
```

Notice the table attributes `mask` and `fill_value` that are available for a masked table.

Convert an existing table to a masked table

```
>>> t = Table([[1, 2], ['x', 'y']])  # standard (unmasked) table
>>> t = Table(t, masked=True, copy=False)  # convert to masked table
```

This operation will convert every Column to MaskedColumn and ensure that any subsequently added columns are masked.

Table Access

Nearly all of the standard methods for accessing and modifying data columns, rows, and individual elements also apply to masked tables.

There are two minor differences for the Row object that is obtained by indexing a single row of a table:

- For standard tables, two such rows can be compared for equality, but in masked tables this comparison will produce an exception.

Both of these differences are due to issues in the underlying numpy.ma.MaskedArray implementation.

Masking and Filling

Both the Table and MaskedColumn classes provide attributes and methods to support manipulating tables with missing or invalid data.

Mask

The mask for a column can be viewed and modified via the `mask` attribute:

```
>>> t = Table([(1, 2), (3, 4)], names=('a', 'b'), masked=True)
>>> t['a'].mask = [False, True]  # Modify column mask (boolean array)
>>> t['b'].mask = [True, False]  # Modify column mask (boolean array)
>>> print(t)
 a   b
```

```
--- ---
  1  --
 --   4
```

Masked entries are shown as `--` when the table is printed. You can view the mask directly, either at the column or table level:

```
>>> t['a'].mask
array([False,  True]...)

>>> t.mask
<Table length=2>
  a     b
 bool  bool
----- -----
False  True
 True False
```

To get the indices of masked elements, use an expression like:

```
>>> t['a'].mask.nonzero()[0]
array([1])
```

Filling

The entries which are masked (i.e., missing or invalid) can be replaced with specified fill values. In this case the `MaskedColumn` or masked `Table` will be converted to a standard `Column` or table. Each column in a masked table has a `fill_value` attribute that specifies the default fill value for that column. To perform the actual replacement operation the `filled()` method is called. This takes an optional argument which can override the default column `fill_value` attribute.

```
>>> t['a'].fill_value = -99
>>> t['b'].fill_value = 33

>>> print(t.filled())
 a   b
--- ---
  1  33
-99   4

>>> print(t['a'].filled())
 a
---
  1
-99
```

```
>>> print(t['a'].filled(999))
 a
---
  1
999

>>> print(t.filled(1000))
 a    b
---- ----
   1 1000
1000    4
```

Masking Change in `astropy` 4.0

In `astropy` 4.0 a change was introduced in the behavior of **`Table`** that impacts the handling of masked columns.

Prior to 4.0, in order to include one or more **`MaskedColumn`** columns in a table, it was required that *every* column be masked, even those with no missing or masked data. This was a holdover from the original implementation of **`Table`** that used a `numpy` structured array as the underlying container for the column data. Since `astropy` 1.0, the **`Table`** object is an ordered dictionary of columns (Table Implementation Details) and there is no requirement that column types be homogenous.

Starting with 4.0, a **`Table`** can contain both **`Column`** and **`MaskedColumn`** columns, and by default the column type is determined solely by the data for each column.

The details of this change are discussed in the sections below.

Note

For most applications, even those with masked column data, we now recommend using the default **`Table`** behavior which allows heterogenous column types. This implies creating tables *without* specifying the `masked` keyword argument.

Meaning of the `masked` Table Attribute

The **`Table`** object has a `masked` attribute which determines the table behavior when adding a new column:

- `masked=True` : non-mixin columns or data are always converted to **`MaskedColumn`**, and mixin columns have a `mask` attribute added if necessary.
- `masked=False` : each column is added based on the type or contents of the data.

The behavior associated with the `masked` attribute has *not changed* in version 4.0. What has changed is that from 4.0 onward a table with `masked=False` may contain **`MaskedColumn`** columns.

It is important to recognize that the `masked` attribute for a table does not imply whether any of the column data are actually masked. A table can have `masked=True` but not have any masked elements in any table column. Starting with version 4.0 there are two table properties which give more useful information about masking:

- `has_masked_columns` : table has at least one **`MaskedColumn`** column. This does *not* check if any data values are actually masked.
- `has_masked_values` : table has one or more column data values which are masked. This may be relatively slow for large tables as it requires checking the mask values of each column.

Starting with version 4.0 the term “masked table” should be reserved for the narrow and less-common case of a table created with `masked=True`. In most cases there should be no need worry about “masked” or “unmasked” at the table level, but instead focus on the individual columns.

Auto-upgrade to Masked

Prior to version 4.0, adding a **`MaskedColumn`** or a new row with masked elements to a table with `masked=False` would set `masked=True` and automatically “upgrade” other columns to be masked. In many cases this upgrade of the other columns was unnecessary and an annoyance.

Starting with 4.0, new columns are added using the column type which is appropriate for the data. For instance, if a `numpy` masked array is added, then that will turn into a **`MaskedColumn`**, but no other columns will be affected and the `masked` attribute will remain as `False`.

A commonly-encountered implication of this change is that tables read with **`read`** will *always* have `masked=False`, and only columns with masked values will be **`MaskedColumn`**. Prior to 4.0 if the input table had any masked values then the returned table would have `masked=True` and all **`MaskedColumn`** columns. An example is in the next section.

Recovering the Pre-4.0 Behavior

For code that requires every existing or newly added column to be masked, it is now required to explicitly specify `masked=True` when creating the table. Previously the table would be auto-upgraded to use **`MaskedColumn`** for all columns as soon as the first masked column was added. If the table already exists (e.g., after using **`read`** to read a data file), then you need to make a new table:

```
>> dat = Table.read('data.fits')
>> dat = Table(dat, masked=True, copy=False)  # Convert to masked
table
>> dat['new_column'] = [1, 2, 3, 4, 5]  # Will be added as a
MaskedColumn
```

For most applications this should not be necessary, and the preferred idiom is the more explicit version below:

```
>> dat = Table.read('data.fits')
>> dat['new_column'] = np.ma.MaskedArray([1, 2, 3, 4, 5])
```

I/O with Tables

Reading and Writing Table Objects

`astropy` provides a unified interface for reading and writing data in different formats. For many common cases this will streamline the process of file I/O and reduce the need to master the separate details of all of the I/O packages within `astropy`. For details and examples of using this interface see the Unified File Read/Write Interface section.

Getting Started

The **`Table`** class includes two methods, **`read()`** and **`write()`**, that make it possible to read from and write to files. A number of formats are automatically supported (see Built-In Table Readers/Writers) and new file formats and extensions can be registered with the **`Table`** class (see I/O Registry (astropy.io.registry)).

To use this interface, first import the **`Table`** class, then call the **`Table`** **`read()`** method with the name of the file and the file format, for instance `'ascii.daophot'`:

```
>>> from astropy.table import Table
>>> t = Table.read('photometry.dat', format='ascii.daophot')
```

It is possible to load tables directly from the Internet using URLs. For example, download tables from VizieR catalogs in CDS format (`'ascii.cds'`):

```
>>> t = Table.read("ftp://cdsarc.u-strasbg.fr/pub/cats/VII/253
/snrs.dat",
...         readme="ftp://cdsarc.u-strasbg.fr/pub/cats/VII/253
/ReadMe",
...          format="ascii.cds")
```

For certain file formats, the format can be automatically detected, for example from the filename extension:

```
>>> t = Table.read('table.tex')
```

Similarly, for writing, the format can be explicitly specified:

```
>>> t.write(filename, format='latex')
```

As for the **`read()`** method, the format may be automatically identified in some cases.

Any additional arguments specified will depend on the format. For examples of this see the section Built-In Table Readers/Writers. This section also provides the full list of choices for the `format` argument.

Supported Formats

The Unified File Read/Write Interface has built-in support for the following data file formats:

- ASCII Formats
- HDF5
- FITS
- VO Tables

Interfacing with the Pandas Package

The pandas package is a package for high performance data analysis of table-like structures that is complementary to the **`Table`** class in `astropy`.

In order to exchange data between the **`Table`** class and the pandas DataFrame class (the main data structure in pandas), the **`Table`** class includes two methods, **`to_pandas()`** and **`from_pandas()`**.

Example

To demonstrate, we can create a minimal table:

```
>>> from astropy.table import Table
>>> t = Table()
>>> t['a'] = [1, 2, 3, 4]
>>> t['b'] = ['a', 'b', 'c', 'd']
```

Which we can then convert to a `pandas` DataFrame:

```
>>> df = t.to_pandas()
>>> df
   a  b
0  1  a
1  2  b
2  3  c
3  4  d
>>> type(df)
<class 'pandas.core.frame.DataFrame'>
```

It is also possible to create a table from a DataFrame:

```
>>> t2 = Table.from_pandas(df)
>>> t2
<Table length=4>
  a      b
int64 string8
----- -------
    1       a
    2       b
    3       c
    4       d
```

The conversions to and from `pandas` are subject to the following caveats:

- The `pandas` DataFrame structure does not support multidimensional columns, so **`Table`** objects with multidimensional columns cannot be converted to DataFrame.
- Masked tables can be converted, but DataFrame uses `numpy.nan` to indicate masked values, so all numerical columns (integer or float) are converted to `numpy.float` columns in DataFrame, and string columns with missing values are converted to object columns with `numpy.nan` values to indicate missing values. For numerical columns, the conversion therefore does not necessarily round-trip if converting back to an `astropy` table, because the distinction between `numpy.nan` and masked values is lost, and the different integer columns (for example) will be converted to floating- point.
- Tables with mixin columns can currently not be converted, but this may be implemented in the future.

Mixin Columns

Mixin Columns

`astropy` tables support the concept of a “mixin column” in tables, which allows integration of appropriate non-**`Column`** based class objects within a **`Table`** object. These mixin column objects are not converted in any way but are used natively.

The available built-in mixin column classes are:

- **`Quantity`** and subclasses
- **`SkyCoord`** and coordinate frame classes
- **`Time`** and **`TimeDelta`**
- **`EarthLocation`**
- **`NdarrayMixin`**

Example

As an example we can create a table and add a time column:

```
>>> from astropy.table import Table
>>> from astropy.time import Time
>>> t = Table()
>>> t['index'] = [1, 2]
>>> t['time'] = Time(['2001-01-02T12:34:56', '2001-02-03T00:01:02'])
>>> print(t)
index           time
----- -----------------------
    1 2001-01-02T12:34:56.000
    2 2001-02-03T00:01:02.000
```

The important point here is that the `time` column is a bona fide **`Time`** object:

```
>>> t['time']
<Time object: scale='utc' format='isot' value=
['2001-01-02T12:34:56.000' '2001-02-03T00:01:02.000']>
>>> t['time'].mjd
array([51911.52425926, 51943.00071759])
```

Quantity and QTable

The ability to natively handle **`Quantity`** objects within a table makes it more convenient to manipulate tabular data with units in a natural and robust way. However, this feature introduces an ambiguity because data with a unit (e.g., from a FITS binary table) can be represented as either a **`Column`** with a `unit` attribute or as a **`Quantity`** object. In order to cleanly resolve this ambiguity, `astropy` defines a minor variant of the **`Table`** class called **`QTable`**. The **`QTable`** class is exactly the same as **`Table`** except that **`Quantity`** is the default for any data column with a defined unit.

If you take advantage of the **Quantity** infrastructure in your analysis, then **QTable** is the preferred way to create tables with units. If instead you use table column units more as a descriptive label, then the plain **Table** class is probably the best class to use.

Example

To illustrate these concepts we first create a standard **Table** where we supply as input a **Time** object and a **Quantity** object with units of `m / s`. In this case the quantity is converted to a **Column** (which has a `unit` attribute but does not have all of the features of a **Quantity**):

```
>>> import astropy.units as u
>>> t = Table()
>>> t['index'] = [1, 2]
>>> t['time'] = Time(['2001-01-02T12:34:56', '2001-02-03T00:01:02'])
>>> t['velocity'] = [3, 4] * u.m / u.s

>>> print(t)
index           time           velocity
                                m / s
----- ----------------------- --------
    1 2001-01-02T12:34:56.000      3.0
    2 2001-02-03T00:01:02.000      4.0

>>> type(t['velocity'])
<class 'astropy.table.column.Column'>

>>> t['velocity'].unit
Unit("m / s")

>>> (t['velocity'] ** 2).unit  # WRONG because Column is not smart
about unit
Unit("m / s")
```

So instead let’s do the same thing using a quantity table **QTable**:

```
>>> from astropy.table import QTable

>>> qt = QTable()
>>> qt['index'] = [1, 2]
>>> qt['time'] = Time(['2001-01-02T12:34:56', '2001-02-03T00:01:02'])
>>> qt['velocity'] = [3, 4] * u.m / u.s
```

The `velocity` column is now a **Quantity** and behaves accordingly:

```
>>> type(qt['velocity'])
<class 'astropy.units.quantity.Quantity'>

>>> qt['velocity'].unit
Unit("m / s")

>>> (qt['velocity'] ** 2).unit  # GOOD!
Unit("m2 / s2")
```

You can conveniently convert **`Table`** to **`QTable`** and vice-versa:

```
>>> qt2 = QTable(t)
>>> type(qt2['velocity'])
<class 'astropy.units.quantity.Quantity'>

>>> t2 = Table(qt2)
>>> type(t2['velocity'])
<class 'astropy.table.column.Column'>
```

Note

To summarize: the **only** difference between **`QTable`** and **`Table`** is the behavior when adding a column that has a specified unit. With **`QTable`** such a column is always converted to a **`Quantity`** object before being added to the table. Likewise if a unit is specified for an existing unit-less **`Column`** in a **`QTable`**, then the column is converted to **`Quantity`**.

The converse is that if you add a **`Quantity`** column to an ordinary **`Table`** then it gets converted to an ordinary **`Column`** with the corresponding `unit` attribute.

Mixin Attributes

The usual column attributes `name`, `dtype`, `unit`, `format`, and `description` are available in any mixin column via the `info` property:

```
>>> qt['velocity'].info.name
'velocity'
```

This `info` property is a key bit of glue that allows a non-Column object to behave much like a column.

The same `info` property is also available in standard **`Column`** objects. These `info` attributes like `t['a'].info.name` refer to the direct **`Column`** attribute (e.g., `t['a'].name`) and can be used interchangeably. Likewise in a **`Quantity`** object, `info.dtype` attribute refers to the native `dtype` attribute of the object.

Note

When writing generalized code that handles column objects which might be mixin columns, you must *always* use the `info` property to access column attributes.

Details and Caveats

Most common table operations behave as expected when mixin columns are part of the table. However, there are limitations in the current implementation.

Adding or inserting a row

Adding or inserting a row works as expected only for mixin classes that are mutable (data can be changed internally) and that have an `insert()` method. **Quantity** and **Time** support `insert()` but, for example, **SkyCoord** does not. If you tried to insert a row into a table with a **SkyCoord** column then an exception like the following would occur:

```
ValueError: Unable to insert row because of exception in column
'skycoord':
'SkyCoord' object has no attribute 'insert'
```

Initializing from a list of rows or a list of dicts

This mode of initializing a table does not work with mixin columns, so both of the following will fail:

```
>>> qt = QTable([{'a': 1 * u.m, 'b': 2},
...               {'a': 2 * u.m, 'b': 3}])
Traceback (most recent call last):
 ...
TypeError: only dimensionless scalar quantities can be converted to
Python scalars

>>> qt = QTable(rows=[[1 * u.m, 2],
...                   [2 * u.m, 3]])
Traceback (most recent call last):
 ...
TypeError: only dimensionless scalar quantities can be converted to
Python scalars
```

The problem lies in knowing if and how to assemble the individual elements for each column into an appropriate mixin column. The current code uses `numpy` to perform this function on numerical or string types, but it does not handle mixin column types like **Quantity** or **SkyCoord**.

Masking

Mixin columns do not generally support masking (with the exception of `Time`), but there is limited support for use of mixins within a masked table. In this case a `mask` attribute is assigned to the mixin column object. This `mask` is a special object that is a boolean array of `False` corresponding to the mixin data shape. The `mask` looks like a normal `numpy` array but an exception will be raised if `True` is assigned to any element. The consequences of the limitation are most apparent in the high-level table operations.

High-level table operations

The table below gives a summary of support for high-level operations on tables that contain mixin columns:

Operation	Support
Grouped Operations	Not implemented yet, but no fundamental limitation.
Stack Vertically	Available for `Quantity` subclasses, `Time` and any other mixin classes that provide a new_like() method in the `info` descriptor.
Stack Horizontally	Works if output mixin column supports masking or if no masking is required.
Join	Works if output mixin column supports masking or if no masking is required; key columns must be subclasses of `numpy.ndarray`.
Unique Rows	Not implemented yet, uses grouped operations.

ASCII table writing

Tables with mixin columns can be written out to file using the `astropy.io.ascii` module, but the fast C-based writers are not available. Instead, the pure-Python writers will be used. For writing tables with mixin columns it is recommended to use the `'ecsv'` ASCII format. This will fully serialize the table data and metadata, allowing full “round-trip” of the table when it is read back. See ECSV Format for details.

Binary table writing

Starting with `` `astropy `` 3.0, tables with mixin columns can be written in binary format to file using both FITS and HDF5 formats. These can be read back to recover exactly the original `Table` including mixin columns and metadata. See Unified File Read/Write Interface for details.

Mixin Protocol

A key idea behind mixin columns is that any class which satisfies a specified protocol can be used. That means many user-defined class objects which handle array-like data can be used natively within a `Table`. The protocol is relatively concise and requires that a class behave like a minimal `numpy` array with the following properties:

- Contains array-like data.
- Implements `__getitem__` to support getting data as a single item, slicing, or index array access.
- Has a `shape` attribute.
- Has a `__len__` method for length.
- Has an `info` class descriptor which is a subclass of the `astropy.utils.data_info.MixinInfo` class.

The Example: ArrayWrapper section shows a minimal working example of a class which can be used as a mixin column. A pandas.Series object can function as a mixin column as well.

Other interesting possibilities for mixin columns include:

- Columns which are dynamically computed as a function of other columns (AKA spreadsheet).
- Columns which are themselves a **`Table`** (i.e., nested tables). A proof of concept is available.

new_like() method

In order to support high-level operations like **`join`** and **`vstack`**, a mixin class must provide a `new_like()` method in the `info` class descriptor. A key part of the functionality is to ensure that the input column metadata are merged appropriately and that the columns have consistent properties such as the shape.

A mixin class that provides `new_like()` must also implement `__setitem__` to support setting via a single item, slicing, or index array.

The `new_like` method has the following signature:

```
def new_like(self, cols, length, metadata_conflicts='warn',
name=None):
    """
    Return a new instance of this class which is consistent with the
    input ``cols`` and has ``length`` rows.

    This is intended for creating an empty column object whose
elements can
    be set in-place for table operations like join or vstack.

    Parameters
    ----------
    cols : list
        List of input columns
    length : int
        Length of the output column object
    metadata_conflicts : str ('warn'|'error'|'silent')
        How to handle metadata conflicts
    name : str
        Output column name

    Returns
    -------
    col : object
        New instance of this class consistent with ``cols``
    """
```

Examples of this are found in the `ColumnInfo` and `QuantityInfo` classes.

Example: ArrayWrapper

The code listing below shows an example of a data container class which acts as a mixin column class. This class is a wrapper around a `numpy` array. It is used in the `astropy` mixin test suite and is fully compliant as a mixin column.

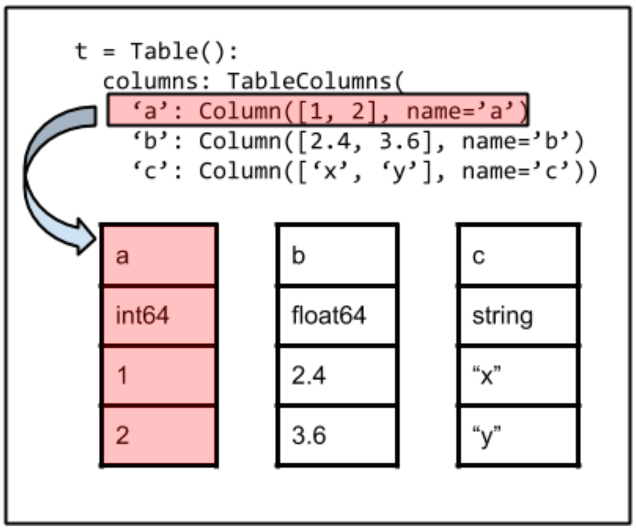
```
from astropy.utils.data_info import ParentDtypeInfo

class ArrayWrapper(object):
    """
    Minimal mixin using a simple wrapper around a numpy array
    """
    info = ParentDtypeInfo()

    def __init__(self, data):
        self.data = np.array(data)
        if 'info' in getattr(data, '__dict__', ()):
            self.info = data.info

    def __getitem__(self, item):
        if isinstance(item, (int, np.integer)):
            out = self.data[item]
        else:
            out = self.__class__(self.data[item])
            if 'info' in self.__dict__:
                out.info = self.info
        return out

    def __setitem__(self, item, value):
        self.data[item] = value

    def __len__(self):
        return len(self.data)

    @property
    def dtype(self):
            return self.data.dtype

    @property
    def shape(self):
        return self.data.shape

    def __repr__(self):
        return ("<{0} name='{1}' data={2}>"
                .format(self.__class__.__name__, self.info.name,
self.data))
```

Implementation

Table Implementation Details

This page provides a brief overview of the **Table** class implementation, in particular highlighting the internal data storage architecture. This is aimed at developers and/or users who are interested in optimal use of the **Table** class.

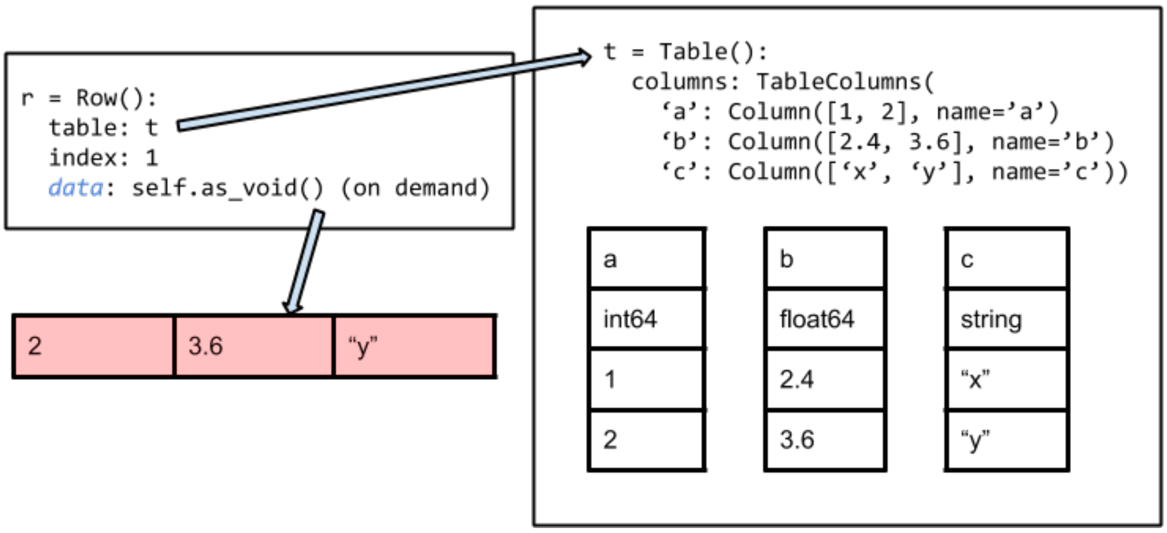
The image below illustrates the basic architecture of the **Table** class. The fundamental data container is an ordered dictionary of individual column objects maintained as the `columns` attribute. It is via this container that columns are managed and accessed.


t = Table():
columns: TableColumns(
'a': Column([1, 2], name='a')
'b': Column([2.4, 3.6], name='b')
'c': Column(['x', 'y'], name='c'))
a
int64
1
2
b
float64
2.4
3.6
c
string
"x"
"y"

Each **Column** (or **MaskedColumn**) object is an **ndarray** subclass and is the sole owner of its data. Maintaining the table as separate columns simplifies table management considerably. It also makes operations like adding or removing columns much faster in comparison to implementations using a `numpy` structured array container.

As shown below, a **Row** object corresponds to a single row in the table. The

Row object does not create a view of the full row at any point. Instead it manages access (e.g., `row['a']`) dynamically by referencing the appropriate elements of the parent table.


r = Row():
table: t
index: 1
data: self.as_void() (on demand)
2
3.6
"y"
t = Table():
columns: TableColumns(
'a': Column([1, 2], name='a')
'b': Column([2.4, 3.6], name='b')
'c': Column(['x', 'y'], name='c'))
a
int64
1
2
b
float64
2.4
3.6
c
string
"x"
"y"

In some cases it is desirable to have a static copy of the full row. This is available via the **as_void()** method, which creates and returns a `numpy.void` or `numpy.ma.mvoid` object with a copy of the original data.

Performance Tips

Constructing **Table** objects row by row using **add_row()** can be very slow:

```
>>> from astropy.table import Table
>>> t = Table(names=['a', 'b'])
>>> for i in range(100):
...    t.add_row((1, 2))
```

If you do need to loop in your code to create the rows, a much faster approach is to construct a list of rows and then create the **Table** object at the very end:

```
>>> rows = []
>>> for i in range(100):
...    rows.append((1, 2))
>>> t = Table(rows=rows, names=['a', 'b'])
```

Writing a **Table** with **MaskedColumn** to `.ecsv` using **write()** can be very slow:

```
>>> from astropy.table import Table
>>> import numpy as np
>>> x = np.arange(10000, dtype=float)
>>> tm = Table([x], masked=True)
>>> tm.write('tm.ecsv', overwrite=True)
```

If you want to write `.ecsv` using **write()**, then use `serialize_method='data_mask'`. This uses the non-masked version of data and it is faster:

```
>>> tm.write('tm.ecsv', overwrite=True, serialize_method='data_mask')
```

Read FITS with memmap=True

By default **read()** will read the whole table into memory, which can take a lot of memory and can take a lot of time, depending on the table size and file format. In some cases, it is possible to only read a subset of the table by choosing the option `memmap=True`.

For FITS binary tables, the data is stored row by row, and it is possible to read only a subset of rows, but reading a full column loads the whole table data into memory:

```
>>> import numpy as np
>>> from astropy.table import Table
>>> tbl = Table({'a': np.arange(1e7),
...              'b': np.arange(1e7, dtype=float),
...              'c': np.arange(1e7, dtype=float)})
>>> tbl.write('test.fits', overwrite=True)
>>> table = Table.read('test.fits', memmap=True)  # Very fast,
doesn't actually load data
>>> table2 = tbl[:100]  # Fast, will read only first 100 rows
>>> print(table2)  # Accessing column data triggers the read
 a    b    c
---- ---- ----
0.0  0.0  0.0
1.0  1.0  1.0
2.0  2.0  2.0
...  ...  ...
98.0 98.0 98.0
99.0 99.0 99.0
Length = 100 rows
>>> col = table['my_column']  # Will load all table into memory
```

At the moment **read()** does not support `memmap=True` for the HDF5 and ASCII file formats.

Reference/API

astropy.table Package

Functions

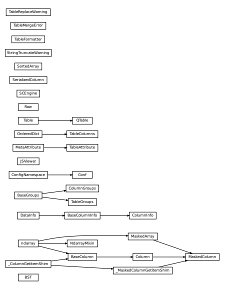
`hstack`(tables[, join_type, uniq_col_name, …])	Stack tables along columns (horizontally)
`join`(left, right[, keys, join_type, …])	Perform a join of the left table with the right table on specified keys.
`represent_mixins_as_columns`(tbl[, …])	Represent input Table `tbl` using only `Column` or `MaskedColumn` objects.
`setdiff`(table1, table2[, keys])	Take a set difference of table rows.
`unique`(input_table[, keys, silent, keep])	Returns the unique rows of a table.
`vstack`(tables[, join_type, metadata_conflicts])	Stack tables vertically (along rows)
`dstack`(tables[, join_type, metadata_conflicts])	Stack columns within tables depth-wise
`join_skycoord`(distance[, distance_func])	Helper function to join on SkyCoord columns using distance matching.
`join_distance`(distance[, kdtree_args, …])	Helper function to join table columns using distance matching.

Classes

`BST`(data, row_index[, unique])	A basic binary search tree in pure Python, used as an engine for indexing.
`Column`([data, name, dtype, shape, length, …])	Define a data column for use in a Table object.
`ColumnGroups`(parent_column[, indices, keys])	
`ColumnInfo`([bound])	Container for meta information like name, description, format.
`Conf`()	Configuration parameters for `astropy.table`.
`JSViewer`([use_local_files, display_length])	Provides an interactive HTML export of a Table.
`MaskedColumn`([data, name, mask, fill_value, …])	Define a masked data column for use in a Table object.
`NdarrayMixin`(obj, *args, **kwargs)	Mixin column class to allow storage of arbitrary numpy ndarrays within a Table.
`QTable`([data, masked, names, dtype, meta, …])	A class to represent tables of heterogeneous data.
`Row`(table, index)	A class to represent one row of a Table object.
`SCEngine`(data, row_index[, unique])	Fast tree-based implementation for indexing, using the `sortedcontainers` package.

SerializedColumn	Subclass of dict that is a used in the representation to contain the name (and possible other info) for a mixin attribute (either primary data or an array-like attribute) that is serialized as a column in the table.
SortedArray(data, row_index[, unique])	Implements a sorted array container using a list of numpy arrays.
StringTruncateWarning	Warning class for when a string column is assigned a value that gets truncated because the base (numpy) string length is too short.
Table([data, masked, names, dtype, meta, …])	A class to represent tables of heterogeneous data.
TableAttribute([default])	Descriptor to define a custom attribute for a Table subclass.
TableColumns([cols])	OrderedDict subclass for a set of columns.
TableFormatter()	
TableGroups(parent_table[, indices, keys])	
TableMergeError	
TableReplaceWarning	Warning class for cases when a table column is replaced via the Table.__setitem__ syntax e.g.

Class Inheritance Diagram



Time and Dates (`astropy.time`)

Introduction

The **`astropy.time`** package provides functionality for manipulating times and dates. Specific emphasis is placed on supporting time scales (e.g., UTC, TAI, UT1, TDB) and time representations (e.g., JD, MJD, ISO 8601) that are used in astronomy and required to calculate, for example, sidereal times and barycentric corrections. The **`astropy.time`** package is based on fast and memory efficient PyERFA wrappers around the ERFA time and calendar routines.

All time manipulations and arithmetic operations are done internally using two 64-bit floats to represent time. Floating point algorithms from [1] are used so that the **Time** object maintains sub-nanosecond precision over times spanning the age of the universe.

[1] Shewchuk, 1997, Discrete & Computational Geometry 18(3):305-363

Getting Started

The usual way to use **astropy.time** is to create a **Time** object by supplying one or more input time values as well as the time format and time scale of those values. The input time(s) can either be a single scalar like `"2010-01-01 00:00:00"` or a list or a `numpy` array of values as shown below. In general, any output values have the same shape (scalar or array) as the input.

Examples

To create a **Time** object:

```
>>> import numpy as np
>>> from astropy.time import Time
>>> times = ['1999-01-01T00:00:00.123456789', '2010-01-01T00:00:00']
>>> t = Time(times, format='isot', scale='utc')
>>> t
<Time object: scale='utc' format='isot' value=
['1999-01-01T00:00:00.123' '2010-01-01T00:00:00.000']>
>>> t[1]
<Time object: scale='utc' format='isot'
value=2010-01-01T00:00:00.000>
```

The `format` argument specifies how to interpret the input values (e.g., ISO, JD, or Unix time). The `scale` argument specifies the time scale for the values (e.g., UTC, TT, or UT1). The `scale` argument is optional and defaults to UTC except for Time from Epoch Formats.

We could have written the above as:

```
>>> t = Time(times, format='isot')
```

When the format of the input can be unambiguously determined, the `format` argument is not required, so we can then simplify even further:

```
>>> t = Time(times)
```

Now we can get the representation of these times in the JD and MJD formats by requesting the corresponding **Time** attributes:

```
>>> t.jd
array([2451179.50000143, 2455197.5       ])
>>> t.mjd
array([51179.00000143, 55197.        ])
```

The full power of output representation is available via the **to_value** method which also allows controlling the subformat. For instance, using `numpy.longdouble` as the output type for higher precision:

```
>>> t.to_value('mjd', 'long')
array([51179.00000143, 55197.        ], dtype=float128)
```

The default representation can be changed by setting the `format` attribute:

```
>>> t.format = 'fits'
>>> t
<Time object: scale='utc' format='fits' value=
['1999-01-01T00:00:00.123'

'2010-01-01T00:00:00.000']>
>>> t.format = 'isot'
```

We can also convert to a different time scale, for instance from UTC to TT. This uses the same attribute mechanism as above but now returns a new **Time** object:

```
>>> t2 = t.tt
>>> t2
<Time object: scale='tt' format='isot' value=
['1999-01-01T00:01:04.307' '2010-01-01T00:01:06.184']>
>>> t2.jd
array([2451179.5007443 , 2455197.50076602])
```

Note that both the ISO (ISOT) and JD representations of `t2` are different than for `t` because they are expressed relative to the TT time scale. Of course, from the numbers or strings you would not be able to tell this was the case:

```
>>> print(t2.fits)
['1999-01-01T00:01:04.307' '2010-01-01T00:01:06.184']
```

You can set the time values in place using the usual `numpy` array setting item syntax:

```
>>> t2 = t.tt.copy()  # Copy required if transformed Time will be
```

```
modified
>>> t2[1] = '2014-12-25'
>>> print(t2)
['1999-01-01T00:01:04.307' '2014-12-25T00:00:00.000']
```

The **`Time`** object also has support for missing values, which is particularly useful for Table Operations such as joining and stacking:

```
>>> t2[0] = np.ma.masked  # Declare that first time is missing or
invalid
>>> print(t2)
[-- '2014-12-25T00:00:00.000']
```

Finally, some further examples of what is possible. For details, see the API documentation below.

```
>>> dt = t[1] - t[0]
>>> dt
<TimeDelta object: scale='tai' format='jd' value=4018.00002172>
```

Here, note the conversion of the timescale to TAI. Time differences can only have scales in which one day is always equal to 86400 seconds.

```
>>> import numpy as np
>>> t[0] + dt * np.linspace(0.,1.,12)
<Time object: scale='utc' format='isot' value=
['1999-01-01T00:00:00.123' '2000-01-01T06:32:43.930'
 '2000-12-31T13:05:27.737' '2001-12-31T19:38:11.544'
 '2003-01-01T02:10:55.351' '2004-01-01T08:43:39.158'
 '2004-12-31T15:16:22.965' '2005-12-31T21:49:06.772'
 '2007-01-01T04:21:49.579' '2008-01-01T10:54:33.386'
 '2008-12-31T17:27:17.193' '2010-01-01T00:00:00.000']>
```

```
>>> t.sidereal_time('apparent', 'greenwich')
<Longitude [6.68050179, 6.70281947] hourangle>
```

Using `astropy.time`

Time Object Basics

In **`astropy.time`** a “time” is a single instant of time which is independent of the way the time is represented (the “format”) and the time “scale” which specifies the offset and scaling relation of the unit of time. There is no distinction made between a “date” and a “time” since both concepts (as loosely defined in common usage) are just different representations of a moment in time.

Time Format

The time format specifies how an instant of time is represented. The currently available formats are can be found in the `Time.FORMATS` dict and are listed in the table below. Each of these formats is implemented as a class that derives from the base **`TimeFormat`** class. This class structure can be adapted and extended by users for specialized time formats not supplied in **`astropy.time`**.

Format	Class	Example Argument
byear	`TimeBesselianEpoch`	1950.0
byear_str	`TimeBesselianEpochString`	'B1950.0'
cxcsec	`TimeCxcSec`	63072064.184
datetime	`TimeDatetime`	datetime(2000, 1, 2, 12, 0, 0)
decimalyear	`TimeDecimalYear`	2000.45
fits	`TimeFITS`	'2000-01-01T00:00:00.000'
gps	`TimeGPS`	630720013.0
iso	`TimeISO`	'2000-01-01 00:00:00.000'
isot	`TimeISOT`	'2000-01-01T00:00:00.000'
jd	`TimeJD`	2451544.5
jyear	`TimeJulianEpoch`	2000.0
jyear_str	`TimeJulianEpochString`	'J2000.0'
mjd	`TimeMJD`	51544.0
plot_date	`TimePlotDate`	730120.0003703703
unix	`TimeUnix`	946684800.0
unix_tai	`TimeUnixTai`	946684800.0
yday	`TimeYearDayTime`	2000:001:00:00:00.000
ymdhms	`TimeYMDHMS`	{'year': 2010, 'month': 3, 'day': 1}
datetime64	`TimeDatetime64`	np.datetime64('2000-01-01T01:01:01')

Note

The **`TimeFITS`** format implements most of the FITS standard [2], including support for the `LOCAL` timescale. Note, though, that FITS supports some deprecated names for timescales; these are translated to the formal names upon initialization. Furthermore, any specific realization information, such as `UT(NIST)` is stored only as long as the time scale is not changed.

[2] Rots et al. 2015, A&A 574:A36

Changing Format

The default representation can be changed by setting the `format` attribute:

```
>>> t = Time('2000-01-02')
>>> t.format = 'jd'
>>> t
<Time object: scale='utc' format='jd' value=2451545.5>
```

Be aware that when changing format, the current output subformat (see section below) may not exist in the new format. In this case, the subformat will not be preserved:

```
>>> t = Time('2000-01-02', format='fits', out_subfmt='longdate')
>>> t.value
'+02000-01-02'
>>> t.format = 'iso'
>>> t.out_subfmt
u'*'
>>> t.format = 'fits'
>>> t.value
'2000-01-02T00:00:00.000'
```

Subformat

Many of the available time format classes support the concept of a subformat. This allows for variations on the basic theme of a format in both the input parsing/validation and the output.

The table below illustrates available subformats for the string formats `iso`, `fits`, and `yday` formats:

Format	Subformat	Input / Output
`iso`	date_hms	2001-01-02 03:04:05.678
`iso`	date_hm	2001-01-02 03:04
`iso`	date	2001-01-02
`fits`	date_hms	2001-01-02T03:04:05.678
`fits`	longdate_hms	+02001-01-02T03:04:05.678

Format	Subformat	Input / Output
`fits`	longdate	+02001-01-02
`yday`	date_hms	2001:032:03:04:05.678
`yday`	date_hm	2001:032:03:04
`yday`	date	2001:032

Numerical formats such as `mjd`, `jyear`, or `cxcsec` all support the subformats: `'float'`, `'long'`, `'decimal'`, `'str'`, and `'bytes'`. Here, `'long'` uses `numpy.longdouble` for somewhat enhanced precision (with the enhancement depending on platform), and `'decimal'` instances of **`decimal.Decimal`** for full precision. For the `'str'` and `'bytes'` subformats, the number of digits is also chosen such that time values are represented accurately.

When used on input, these formats allow creating a time using a single input value that accurately captures the value to the full available precision in **`Time`**. Conversely, the single value on output using **`Time to_value`** or **`TimeDelta to_value`** can have higher precision than the standard 64-bit float:

```
>>> tm = Time('51544.000000000000001', format='mjd')  # String input
>>> tm.mjd  # float64 output loses last digit but Decimal gets it
51544.0
>>> tm.to_value('mjd', subfmt='decimal')
Decimal('51544.00000000000000099920072216264')
>>> tm.to_value('mjd', subfmt='str')
'51544.000000000000001'
```

The complete list of subformat options for the **`Time`** formats that have them is:

Format	Subformats
`byear`	float, long, decimal, str, bytes
`cxcsec`	float, long, decimal, str, bytes
`datetime64`	date_hms, date_hm, date
`decimalyear`	float, long, decimal, str, bytes
`fits`	date_hms, date, longdate_hms, longdate
`gps`	float, long, decimal, str, bytes
`iso`	date_hms, date_hm, date
`isot`	date_hms, date_hm, date
`jd`	float, long, decimal, str, bytes

Format	Subformats
`jyear`	float, long, decimal, str, bytes
`mjd`	float, long, decimal, str, bytes
`plot_date`	float, long, decimal, str, bytes
`unix`	float, long, decimal, str, bytes
`unix_tai`	float, long, decimal, str, bytes
`yday`	date_hms, date_hm, date

The complete list of subformat options for the **`TimeDelta`** formats that have them is:

Format	Subformats
`jd`	float, long, decimal, str, bytes
`sec`	float, long, decimal, str, bytes

Time from Epoch Formats

The formats `cxcsec`, `gps`, `unix`, and `unix_tai` are special in that they provide a floating point representation of the elapsed time in seconds since a particular reference date. These formats have a intrinsic time scale which is used to compute the elapsed seconds since the reference date.

Format	Scale	Reference date
`cxcsec`	TT	`1998-01-01 00:00:00`
`unix`	UTC	`1970-01-01 00:00:00`
`unix_tai`	TAI	`1970-01-01 00:00:08`
`gps`	TAI	`1980-01-06 00:00:19`

Unlike the other formats which default to UTC, if no `scale` is provided when initializing a **`Time`** object then the above intrinsic scale is used. This is done for computational efficiency.

Time Scale

The time scale (or time standard) is “a specification for measuring time: either the rate at which time passes; or points in time; or both” [3], [4].

```
>>> Time.SCALES
('tai', 'tcb', 'tcg', 'tdb', 'tt', 'ut1', 'utc', 'local')
```

Scale	Description
tai	International Atomic Time (TAI)
tcb	Barycentric Coordinate Time (TCB)
tcg	Geocentric Coordinate Time (TCG)
tdb	Barycentric Dynamical Time (TDB)
tt	Terrestrial Time (TT)
ut1	Universal Time (UT1)
utc	Coordinated Universal Time (UTC)
local	Local Time Scale (LOCAL)

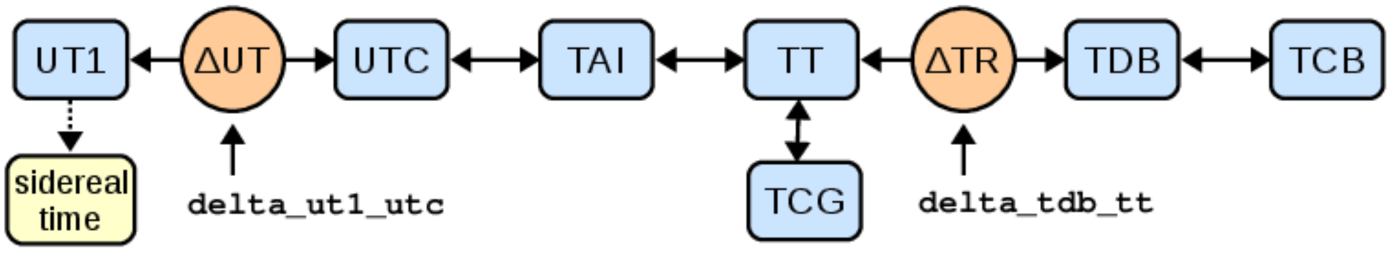
[3] Wikipedia time standard article

[4] SOFA Time Scale and Calendar Tools (PDF)

Note

The `local` time scale is meant for free-running clocks or simulation times (i.e., to represent a time without a properly defined scale). This means it cannot be converted to any other time scale, and arithmetic is possible only with **`Time`** instances with scale `local` and with **`TimeDelta`** instances with scale `local` or **`None`**.

The system of transformation between supported time scales (i.e., all but `local`) is shown in the figure below. Further details are provided in the Convert time scale section.



Scalar or Array

A **`Time`** object can hold either a single time value or an array of time values. The distinction is made entirely by the form of the input time(s). If a **`Time`** object holds a single value then any format outputs will be a single scalar value, and likewise for arrays.

Example

Like other arrays and lists, `Time` objects holding arrays are subscriptable, returning scalar or array objects as appropriate:

```
>>> from astropy.time import Time
>>> t = Time(100.0, format='mjd')
>>> t.jd
2400100.5
>>> t = Time([100.0, 200.0, 300.], format='mjd')
>>> t.jd
array([2400100.5, 2400200.5, 2400300.5])
>>> t[:2]
<Time object: scale='utc' format='mjd' value=[100. 200.]>
>>> t[2]
<Time object: scale='utc' format='mjd' value=300.0>
>>> t = Time(np.arange(50000., 50003.)[:, np.newaxis],
...          np.arange(0., 1., 0.5), format='mjd')
>>> t
<Time object: scale='utc' format='mjd' value=[[50000.  50000.5]
 [50001.  50001.5]
 [50002.  50002.5]]>
>>> t[0]
<Time object: scale='utc' format='mjd' value=[50000.  50000.5]>
```

NumPy Method Analogs and Applicable NumPy Functions

For `Time` instances holding arrays, many of the same methods and attributes that work on `ndarray` instances can be used. For example, you can reshape `Time` instances and take specific parts using `reshape()`, `ravel()`, `flatten()`, `T`, `transpose()`, `swapaxes()`, `diagonal()`, `squeeze()`, or `take()`. Similarly, on `numpy` version 1.17 and later corresponding functions as well as others that affect the shape, such as `atleast_1d` and `rollaxis`, work as expected. (The relevant functions have to be explicitly enabled in `astropy` source code; let us know if a `numpy` function is not supported that you think should work.)

Examples

To reshape `Time` instances:

```
.. doctest-requires:: numpy>=1.17
```

```
>>> t.reshape(2, 3)
<Time object: scale='utc' format='mjd' value=[[50000.   50000.5 50001. ]
 [50001.5 50002.  50002.5]]>
>>> t.T
<Time object: scale='utc' format='mjd' value=[[50000.   50001.   50002. ]
 [50000.5 50001.5 50002.5]]>
>>> np.roll(t, 1, axis=0)
<Time object: scale='utc' format='mjd' value=[[50002.   50002.5]
 [50000.   50000.5]
 [50001.   50001.5]]>
```

Note that similarly to the **ndarray** methods, all but **flatten()** try to use new views of the data, with the data copied only if that is impossible (as discussed, for example, in the documentation for `numpy` **reshape()**).

Some arithmetic methods are supported as well: **min()**, **max()**, **ptp()**, **sort()**, **argmin()**, **argmax()**, and **argsort()**.

To apply arithmetic methods to **Time** instances:

```
>> t.max()
<Time object: scale='utc' format='mjd' value=50002.5>
>> t.ptp(axis=0)  # doctest: +FLOAT_CMP
<TimeDelta object: scale='tai' format='jd' value=[2. 2.]>
```

Inferring Input Format

The **Time** class initializer will not accept ambiguous inputs, but it will make automatic inferences in cases where the inputs are unambiguous. This can apply when the times are supplied as objects, inputs for `ymdhms`, or strings. In the latter case it is not required to specify the format because the available string formats have no overlap. However, if the format is known in advance the string parsing will be faster if the format is provided.

Example
To infer input format:

```
>>> from datetime import datetime
>>> t = Time(datetime(2010, 1, 2, 1, 2, 3))
>>> t.format
'datetime'
>>> t = Time('2010-01-02 01:02:03')
>>> t.format
'iso'
```

Internal Representation

The **`Time`** object maintains an internal representation of time as a pair of double precision numbers expressing Julian days. The sum of the two numbers is the Julian Date for that time relative to the given time scale. Users requiring no better than microsecond precision over human time scales (~100 years) can safely ignore the internal representation details and skip this section.

This representation is driven by the underlying ERFA C-library implementation. The ERFA routines take care throughout to maintain overall precision of the double pair. Users are free to choose the way in which total JD is provided, though internally one part contains integer days and the other the fraction of the day, as this ensures optimal accuracy for all conversions. The internal JD pair is available via the `jd1` and `jd2` attributes:

```
>>> t = Time('2010-01-01 00:00:00', scale='utc')
>>> t.jd1, t.jd2
(2455198.0, -0.5)
>>> t2 = t.tai
>>> t2.jd1, t2.jd2
(2455198., -0.49960648148148146)
```

Creating a Time Object

The allowed **`Time`** arguments to create a time object are listed below:

val : *numpy ndarray, list, str, or number*
  Data to initialize table.

val2 : *numpy ndarray, list, str, or number; optional*
  Data to initialize table.

format : *str, optional*
  Format of input value(s).

scale : *str, optional*
  Time scale of input value(s).

precision : *int between 0 and 9 inclusive*

Decimal precision when outputting seconds as floating point.

in_subfmt : *str*

Unix glob to select subformats for parsing input times.

out_subfmt : *str*

Unix glob to select subformat for output times.

location : `EarthLocation` *or tuple, optional*

If a tuple, three `Quantity` items with length units for geocentric coordinates, or a longitude, latitude, and optional height for geodetic coordinates. Can be a single location, or one for each input time.

val

The `val` argument specifies the input time or times and can be a single string or number, or it can be a Python list or `` `numpy `` array of strings or numbers. To initialize a `Time` object based on a specified time, it *must* be present.

In most situations, you also need to specify the time scale via the `scale` argument. The `Time` class will never guess the time scale, so a concise example would be:

```
>>> t1 = Time(50100.0, scale='tt', format='mjd')
>>> t2 = Time('2010-01-01 00:00:00', scale='utc')
```

It is possible to create a new `Time` object from one or more existing time objects. In this case, the format and scale will be inferred from the first object unless explicitly specified.

```
>>> Time([t1, t2])
<Time object: scale='tt' format='mjd' value=[50100. 55197.00076602]>
```

val2

The `val2` argument is available for those situations where high precision is required. Recall that the internal representation of time within `astropy.time` is two double-precision numbers that when summed give the Julian date. If provided, the `val2` argument is used in combination with `val` to set the second of the internal time values. The exact interpretation of `val2` is determined by the input format class. All string-valued formats ignore `val2` and all numeric inputs effectively add the two values in a way that maintains the highest precision. For example:

```
>>> t = Time(100.0, 0.000001, format='mjd', scale='tt')
>>> t.jd, t.jd1, t.jd2
(2400100.500001, 2400101.0, -0.499999)
```

format

The `` `format`` argument sets the time time format, and as mentioned it is required unless the format can be unambiguously determined from the input times.

scale

The `scale` argument sets the time scale and is required except for time formats such as `plot_date` (**`TimePlotDate`**) and `unix` (**`TimeUnix`**). These formats represent the duration in SI seconds since a fixed instant in time is independent of time scale. See the Time from Epoch Formats for more details.

precision

The `precision` setting affects string formats when outputting a value that includes seconds. It must be an integer between 0 and 9. There is no effect when inputting time values from strings. The default precision is 3. Note that the limit of 9 digits is driven by the way that ERFA handles fractional seconds. In practice this should should not be an issue.

```
>>> t = Time('B1950.0', precision=3)
>>> t.byear_str
'B1950.000'
>>> t.precision = 0
>>> t.byear_str
'B1950'
```

in_subfmt

The `in_subfmt` argument provides a mechanism to select one or more subformat values from the available subformats for input. Multiple allowed subformats can be selected using Unix-style wildcard characters, in particular `*` and `?`, as documented in the Python fnmatch module.

The default value for `in_subfmt` is `*` which matches any available subformat. This allows for convenient input of values with unknown or heterogeneous subformat:

```
>>> Time(['2000:001', '2000:002:03:04', '2001:003:04:05:06.789'])
<Time object: scale='utc' format='yday'
 value=['2000:001:00:00:00.000' '2000:002:03:04:00.000'
'2001:003:04:05:06.789']>
```

You can explicitly specify `in_subfmt` in order to strictly require a certain subformat:

```
>>> t = Time('2000:002:03:04', in_subfmt='date_hm')
>>> t = Time('2000:002', in_subfmt='date_hm')
Traceback (most recent call last):
  ...
ValueError: Input values did not match any of the formats where the
format keyword is optional ['astropy_time', 'datetime',
'byear_str', 'iso', 'isot', 'jyear_str', 'yday']
```

out_subfmt

The `out_subfmt` argument is similar to `in_subfmt` except that it applies to output formatting. In the case of multiple matching subformats, the first matching subformat is used.

```
>>> Time('2000-01-01 02:03:04', out_subfmt='date').iso
'2000-01-01'
>>> Time('2000-01-01 02:03:04', out_subfmt='date_hms').iso
'2000-01-01 02:03:04.000'
>>> Time('2000-01-01 02:03:04', out_subfmt='date*').iso
'2000-01-01 02:03:04.000'
>>> Time('50814.123456789012345', format='mjd', out_subfmt='str').mjd
'50814.123456789012345'
```

See also the subformat section.

location

This optional parameter specifies the observer location, using an **`EarthLocation`** object or a tuple containing any form that can initialize one: either a tuple with geocentric coordinates (X, Y, Z), or a tuple with geodetic coordinates (longitude, latitude, height; with height defaulting to zero). They are used for time scales that are sensitive to observer location (currently, only TDB, which relies on the PyERFA routine **`erfa.dtdb`** to determine the time offset between TDB and TT), as well as for sidereal time if no explicit longitude is given.

```
>>> t = Time('2001-03-22 00:01:44.732327132980', scale='utc',
...           location=('120d', '40d'))
>>> t.sidereal_time('apparent', 'greenwich')
<Longitude 12. hourangle>
>>> t.sidereal_time('apparent')
<Longitude 20. hourangle>
```

Note

In future versions, we hope to add the possibility to add observatory objects and/or names.

Getting the Current Time

The current time can be determined as a **`Time`** object using the **`now`** class method:

```
>>> nt = Time.now()
>>> ut = Time(datetime.utcnow(), scale='utc')
```

The two should be very close to each other.

Fast C-based Date String Parser

Time formats that are based on a date string representation of time, including **`TimeISO`**, **`TimeISOT`**, and **`TimeYearDayTime`**, make use of a fast C-based date parser that improves speed by a factor of 20 or more for large arrays of times.

The C parser is stricter than the Python-based parser (which relies on `strptime`). In particular fields like the month or day of year must always have a fixed number of ASCII digits. As an example the Python parser will accept `2000-1-2T3:04:5.23` while the C parser requires `2000-01-02T03:04:05.23`

Use of the C parser is enabled by default except when the input subformat `in_subfmt` argument is different from the default value of `'*'`. If the fast C parser fails to parse the date values then the `Time` initializer will automatically fall through to the Python parser.

In rare cases where you need to explicitly control which parser gets used there is a configuration item `time.conf.use_fast_parser` that can be set. The default is `'True'`, which means to try the fast parser and fall through to Python parser if needed. Note that the configuration value is a string, not a bool object.

For example to disable the C parser use:

```
>>> from astropy.time import conf
>>> date = '2000-1-2T3:04:5.23'
>>> t = Time(date, format='isot')  # Succeeds by default
>>> with conf.set_temp('use_fast_parser', 'False'):
...     t = Time(date, format='isot')
...     print(t)
2000-01-02T03:04:05.230
```

To force the user of the C parser (for example in testing) use:

```
>>> with conf.set_temp('use_fast_parser', 'force'):
...     try:
...         t = Time(date, format='isot')
...     except ValueError as err:
...         print(err)
Input values did not match the format class isot:
ValueError: fast C time string parser failed: non-digit found where
digit (0-9) required
```

Using Time Objects

The operations available with `Time` objects include:

- Get and set time value(s) for an array-valued `Time` object.
- Set missing (masked) values.
- Get the representation of the time value(s) in a particular time format.
- Get a new time object for the same time value(s) but referenced to a different time scale.
- Calculate the sidereal time corresponding to the time value(s).

- Do time arithmetic involving `Time` and/or `TimeDelta` objects.

Get and Set Values

For an existing `Time` object which is array-valued, you can use the usual `numpy` array item syntax to get either a single item or a subset of items. The returned value is a `Time` object with all the same attributes.

Examples

To get an item or a subset of items:

```
>>> t = Time(['2001:020', '2001:040', '2001:060', '2001:080'],
...          out_subfmt='date')
>>> print(t[1])
2001:040
>>> print(t[1:])
['2001:040' '2001:060' '2001:080']
>>> print(t[[2, 0]])
['2001:060' '2001:020']
```

You can also set values in place for an array-valued `Time` object:

```
>>> t = Time(['2001:020', '2001:040', '2001:060', '2001:080'],
...          out_subfmt='date')
>>> t[1] = '2010:001'
>>> print(t)
['2001:020' '2010:001' '2001:060' '2001:080']
>>> t[[2, 0]] = '1990:123'
>>> print(t)
['1990:123' '2010:001' '1990:123' '2001:080']
```

The new value (on the right hand side) when setting can be one of three possibilities:

- Scalar string value or array of string values where each value is in a valid time format that can be automatically parsed and used to create a `Time` object.
- Value or array of values where each value has the same `format` as the `Time` object being set. For instance, a float or `numpy` array of floats for an object with `format='unix'`.
- `Time` object with identical `location` (but `scale` and `format` need not be the same). The right side value will be transformed so the time `scale` matches.

Whenever any item is set, then the internal cache (see Caching) is cleared along with the `delta_tdb_tt` and/or `delta_ut1_utc` transformation offsets, if they have been set.

If it is required that the **`Time`** object be immutable, then set the `writeable` attribute to **`False`**. In this case, attempting to set a value will raise a `ValueError: Time object is read-only`. See the section on Caching for an example.

Missing Values

The **`Time`** and **`TimeDelta`** objects support functionality for marking values as missing or invalid. This is also known as masking, and is especially useful for Table Operations such as joining and stacking.

Example

To set one or more items as missing, assign the special value **`numpy.ma.masked`**:

```
>>> t = Time(['2001:020', '2001:040', '2001:060', '2001:080'],
...          out_subfmt='date')
>>> t[2] = np.ma.masked
>>> print(t)
['2001:020' '2001:040' -- '2001:080']
```

Note

The operation of setting an array element to **`numpy.ma.masked`** (missing) *overwrites* the actual time data and therefore there is no way to recover the original value. In this sense, the **`numpy.ma.masked`** value behaves just like any other valid **`Time`** value when setting. This is similar to how Pandas missing data works, but somewhat different from NumPy masked arrays which maintain a separate mask array and retain the underlying data. In the **`Time`** object the `mask` attribute is read-only and cannot be directly set.

Once one or more values in the object are masked, any operations will propagate those values as masked, and access to format attributes such as `unix` or `value` will return a **`MaskedArray`** object:

```
>>> t.unix
masked_array(data = [979948800.0 981676800.0 -- 985132800.0],
             mask = [False False  True False],
        fill_value = 1e+20)
```

You can view the `mask`, but note that it is read-only and setting the mask is always done by setting the item to **`masked`**.

```
>>> t.mask
array([False, False,  True, False]...)
>>> t[:2] = np.ma.masked
```

Warning

The internal implementation of missing value support is provisional and may change in a subsequent release. This would impact information in the next section. However, the documented API for using missing values with **`Time`** and **`TimeDelta`** objects is stable.

Custom Format Classes and Missing Values

For advanced users who have written a custom time format via a **`TimeFormat`** subclass, it may be necessary to modify your class *if you wish to support missing values*. For applications that do not take advantage of missing values no changes are required.

Missing values in a **`TimeFormat`** subclass object are marked by setting the corresponding entries of the `jd2` attribute to be `numpy.nan` (but this is never done directly by the user). For most array operations and `numpy` functions the `numpy.nan` entries are propagated as expected and all is well. However, this is not always the case, and in particular the ERFA routines do not generally support `numpy.nan` values gracefully.

In cases where `numpy.nan` is not acceptable, format class methods should use the `jd2_filled` property instead of `jd2`. This replaces `numpy.nan` with `0.0`. Since `jd2` is always in the range -1 to +1, substituting `0.0` will allow functions to return “reasonable” values which will then be masked in any subsequent outputs. See the `value` property of the **`TimeDecimalYear`** format for any example.

Get Representation

Instants of time can be represented in different ways, for instance as an ISO-format date string (`'1999-07-23 04:31:00'`) or seconds since 1998.0 (`49091460.0`) or Modified Julian Date (`51382.187451574`).

The representation of a **`Time`** object in a particular format is available by getting the object attribute corresponding to the format name. The list of available format names is in the time format section.

```
>>> t = Time('2010-01-01 00:00:00', format='iso', scale='utc')
>>> t.jd        # JD representation of time in current scale (UTC)
2455197.5
>>> t.iso       # ISO representation of time in current scale (UTC)
'2010-01-01 00:00:00.000'
>>> t.unix      # seconds since 1970.0 (UTC)
1262304000.0
>>> t.datetime  # Representation as datetime.datetime object
datetime.datetime(2010, 1, 1, 0, 0)
```

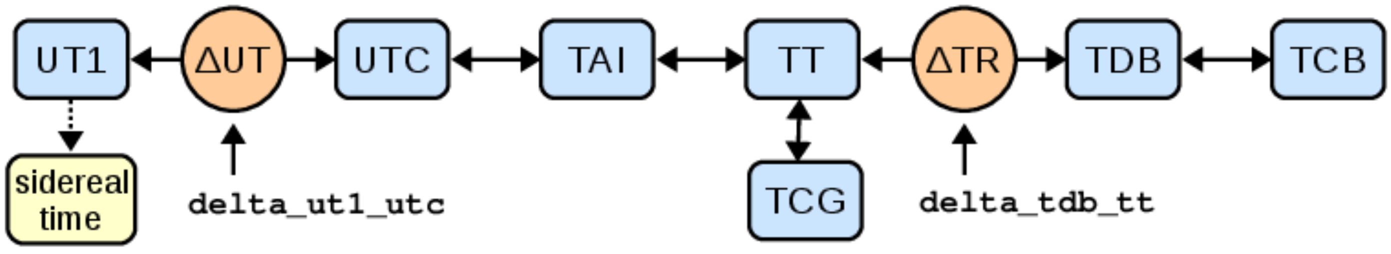
Example

To get the representation of a **`Time`** object:

```
>>> import matplotlib.pyplot as plt
>>> jyear = np.linspace(2000, 2001, 20)
>>> t = Time(jyear, format='jyear')
>>> plt.plot_date(t.plot_date, jyear)
>>> plt.gcf().autofmt_xdate()  # orient date labels at a slant
>>> plt.draw()
```

Convert Time Scale

A new **`Time`** object for the same time value(s) but referenced to a new time scale can be created getting the object attribute corresponding to the time scale name. The list of available time scale names is in the time scale section and in the figure below illustrating the network of time scale transformations.



Examples

To create a **`Time`** object with a new time scale:

```
>>> t = Time('2010-01-01 00:00:00', format='iso', scale='utc')
>>> t.tt        # TT scale
<Time object: scale='tt' format='iso' value=2010-01-01 00:01:06.184>
>>> t.tai
<Time object: scale='tai' format='iso' value=2010-01-01 00:00:34.000>
```

In this process the `format` and other object attributes like `lon`, `lat`, and `precision` are also propagated to the new object.

As noted in the Time Object Basics section, a **`Time`** object can only be changed by explicitly setting some of its elements. The process of changing the time scale therefore begins by making a copy of the original object and then converting the internal time values in the copy to the new time scale. The new **`Time`** object is returned by the attribute access.

Caching

The computations for transforming to different time scales or formats can be time-consuming for large arrays. In order to avoid repeated computations, each **`Time`** or **`TimeDelta`** instance caches such transformations internally:

```
>>> t = Time(np.arange(1e6), format='unix', scale='utc')

>>> time x = t.tt
CPU times: user 263 ms, sys: 4.02 ms, total: 267 ms
Wall time: 267 ms

>>> time x = t.tt
CPU times: user 28 µs, sys: 9 µs, total: 37 µs
Wall time: 32.9 µs
```

Actions such as changing the output precision or subformat will clear the cache. In order to explicitly clear the internal cache do:

```
>>> del t.cache

>>> time x = t.tt
CPU times: user 263 ms, sys: 4.02 ms, total: 267 ms
Wall time: 267 ms
```

In order to ensure consistency between the transformed (and cached) version and the original, the transformed object is set to be not writeable. For example:

```
>>> x = t.tt
>>> x[1] = '2000:001'
Traceback (most recent call last):
  ...
ValueError: Time object is read-only. Make a copy() or set
"writeable" attribute to True.
```

If you require modifying the object then make a copy first, for example, `x = t.tt.copy()`.

Transformation Offsets

Time scale transformations that cross one of the orange circles in the image above require an additional offset time value that is model or observation dependent. See SOFA Time Scale and Calendar Tools for further details.

The two attributes **`delta_ut1_utc`** and **`delta_tdb_tt`** provide a way to set these offset times explicitly. These represent the time scale offsets UT1 - UTC and TDB - TT, respectively. As an example:

```
>>> t = Time('2010-01-01 00:00:00', format='iso', scale='utc')
>>> t.delta_ut1_utc = 0.334  # Explicitly set one part of the
transformation
>>> t.ut1.iso    # ISO representation of time in UT1 scale
'2010-01-01 00:00:00.334'
```

For the UT1 to UTC offset, you have to interpolate the observed values provided by the International Earth Rotation and Reference Systems (IERS) Service. `astropy` will automatically download and use values from the IERS which cover times spanning from 1973-Jan-01 through one year into the future. In addition, the `astropy` package is bundled with a data table of values provided in Bulletin B, which cover the period from 1962 to shortly before an `astropy` release.

When the **`delta_ut1_utc`** attribute is not set explicitly, IERS values will be used (initiating a download of a few Mb file the first time). For details about how IERS values are used in `astropy` time and coordinates, and to understand how to control automatic downloads, see IERS data access (astropy.utils.iers). The example below illustrates converting to the `UT1` scale along with the auto-download feature:

```
>>> t = Time('2016:001')
>>> t.ut1
Downloading https://maia.usno.navy.mil/ser7/finals2000A.all
|==================================================================|
3.0M/3.0M (100.00%)         6s
<Time object: scale='ut1' format='yday' value=2016:001:00:00:00.082>
```

Note

The **IERS_Auto** class contains machinery to ensure that the IERS table is kept up to date by auto-downloading the latest version as needed. This means that the IERS table is assured of having the state-of-the-art definitive and predictive values for Earth rotation. As a user it is **your responsibility** to understand the accuracy of IERS predictions if your science depends on that. If you request `UT1-UTC` for times beyond the range of IERS table data then the nearest available values will be provided.

In the case of the TDB to TT offset, most users need only provide the `lon` and `lat` values when creating the **Time** object. If the **delta_tdb_tt** attribute is not explicitly set, then the PyERFA routine **erfa.dtdb** will be used to compute the TDB to TT offset. Note that if `lon` and `lat` are not explicitly initialized, values of 0.0 degrees for both will be used.

Example

The following code replicates an example in the SOFA Time Scale and Calendar Tools document. It does the transform from UTC to all supported time scales (TAI, TCB, TCG, TDB, TT, UT1, UTC). This requires an observer location (here, latitude and longitude).

```
>>> import astropy.units as u
>>> t = Time('2006-01-15 21:24:37.5', format='iso', scale='utc',
...          location=(-155.933222*u.deg, 19.48125*u.deg))
>>> t.utc.iso
'2006-01-15 21:24:37.500'
>>> t.ut1.iso
'2006-01-15 21:24:37.834'
>>> t.tai.iso
'2006-01-15 21:25:10.500'
>>> t.tt.iso
'2006-01-15 21:25:42.684'
>>> t.tcg.iso
'2006-01-15 21:25:43.323'
>>> t.tdb.iso
'2006-01-15 21:25:42.684'
>>> t.tcb.iso
'2006-01-15 21:25:56.894'
```

Hashing

A user can generate a unique hash key for scalar (0-dimensional) **Time** or

TimeDelta objects. The key is based on a tuple of `jd1`, `jd2`, `scale`, and `location` (if present, `None` otherwise).

Note that two **Time** objects with a different `scale` can compare equally but still have different hash keys. This a practical consideration driven in by performance, but in most cases represents a desirable behavior.

Printing Time Arrays

If your `times` array contains a lot of elements, the `value` argument will display all the elements of the **Time** object `t` when it is called or printed. To control the number of elements to be displayed, set the `threshold` argument with `np.printoptions` as follows:

```
>>> many_times = np.arange(1000)
>>> t = Time(many_times, format='cxcsec')
>>> with np.printoptions(threshold=10):
...     print(repr(t))
...     print(t.iso)
<Time object: scale='tt' format='cxcsec' value=[  0.   1.   2. ...
997. 998. 999.]>
['1998-01-01 00:00:00.000' '1998-01-01 00:00:01.000'
 '1998-01-01 00:00:02.000' ... '1998-01-01 00:16:37.000'
 '1998-01-01 00:16:38.000' '1998-01-01 00:16:39.000']
```

Sidereal Time

Apparent or mean sidereal time can be calculated using **sidereal_time()**. The method returns a **Longitude** with units of hour angle, which by default is for the longitude corresponding to the location with which the **Time** object is initialized. Like the scale transformations, ERFA C-library routines are used under the hood, which support calculations following different IAU resolutions.

Example

To calculate sidereal time:

```
>>> t = Time('2006-01-15 21:24:37.5', scale='utc', location=('120d',
'45d'))
>>> t.sidereal_time('mean')
<Longitude 13.08952187 hourangle>
```

```
>>> t.sidereal_time('apparent')
<Longitude 13.08950368 hourangle>
>>> t.sidereal_time('apparent', 'greenwich')
<Longitude 5.08950368 hourangle>
>>> t.sidereal_time('apparent', '-90d')
<Longitude 23.08950368 hourangle>
>>> t.sidereal_time('apparent', '-90d', 'IAU1994')
<Longitude 23.08950365 hourangle>
```

Time Deltas

Time arithmetic is supported using the `TimeDelta` class. The following operations are available:

- Create a `TimeDelta` explicitly by instantiating a class object.
- Create a `TimeDelta` by subtracting two `Time` objects.
- Add a `TimeDelta` to a `Time` object to get a new `Time`.
- Subtract a `TimeDelta` from a `Time` object to get a new `Time`.
- Add two `TimeDelta` objects to get a new `TimeDelta`.
- Negate a `TimeDelta` or take its absolute value.
- Multiply or divide a `TimeDelta` by a constant or array.
- Convert `TimeDelta` objects to and from time-like `Quantity`’s.

The `TimeDelta` class is derived from the `Time` class and shares many of its properties. One difference is that the time scale has to be one for which one day is exactly 86400 seconds. Hence, the scale cannot be UTC.

The available time formats are:

Format	Class
sec	`TimeDeltaSec`
jd	`TimeDeltaJD`
datetime	`TimeDeltaDatetime`

Examples

Use of the `TimeDelta` object is illustrated in the few examples below:

```
>>> t1 = Time('2010-01-01 00:00:00')
>>> t2 = Time('2010-02-01 00:00:00')
>>> dt = t2 - t1  # Difference between two Times
>>> dt
<TimeDelta object: scale='tai' format='jd' value=31.0>
```

```
>>> dt.sec
2678400.0

>>> from astropy.time import TimeDelta
>>> dt2 = TimeDelta(50.0, format='sec')
>>> t3 = t2 + dt2  # Add a TimeDelta to a Time
>>> t3.iso
'2010-02-01 00:00:50.000'

>>> t2 - dt2  # Subtract a TimeDelta from a Time
<Time object: scale='utc' format='iso' value=2010-01-31 23:59:10.000>

>>> dt + dt2
<TimeDelta object: scale='tai' format='jd' value=31.0005787037>

>>> import numpy as np
>>> t1 + dt * np.linspace(0, 1, 5)
<Time object: scale='utc' format='iso' value=['2010-01-01
00:00:00.000'
'2010-01-08 18:00:00.000' '2010-01-16 12:00:00.000' '2010-01-24
06:00:00.000'
'2010-02-01 00:00:00.000']>
```

The **`TimeDelta`** has a **`to_value`** method which supports controlling the type of the output representation by providing either a format name and optional subformat or a valid `astropy` unit:

```
>>> dt.to_value(u.hr)
744.0
>>> dt.to_value('jd', 'str')
'31.0'
```

Time Scales for Time Deltas

We have shown in the above that the difference between two UTC times is a **`TimeDelta`** with a scale of TAI. This is because a UTC time difference cannot be uniquely defined unless the user knows the two times that were differenced (because of leap seconds, a day does not always have 86400 seconds). For all other time scales, the **`TimeDelta`** inherits the scale of the first **`Time`** object.

Examples

To get the time scale for a **`TimeDelta`** object:

```
>>> t1 = Time('2010-01-01 00:00:00', scale='tcg')
>>> t2 = Time('2011-01-01 00:00:00', scale='tcg')
>>> dt = t2 - t1
>>> dt
<TimeDelta object: scale='tcg' format='jd' value=365.0>
```

When **TimeDelta** objects are added or subtracted from **Time** objects, scales are converted appropriately, with the final scale being that of the **Time** object:

```
>>> t2 + dt
<Time object: scale='tcg' format='iso' value=2012-01-01 00:00:00.000>
>>> t2.tai
<Time object: scale='tai' format='iso' value=2010-12-31 23:59:27.068>
>>> t2.tai + dt
<Time object: scale='tai' format='iso' value=2011-12-31 23:59:27.046>
```

TimeDelta objects can be converted only to objects with compatible scales (i.e., scales for which it is not necessary to know the times that were differenced):

```
>>> dt.tt
<TimeDelta object: scale='tt' format='jd' value=364.999999746>
>>> dt.tdb
Traceback (most recent call last):
  ...
ScaleValueError: Cannot convert TimeDelta with scale 'tcg' to scale
'tdb'
```

TimeDelta objects can also have an undefined scale, in which case it is assumed that their scale matches that of the other **Time** or **TimeDelta** object (or is TAI in case of a UTC time):

```
>>> t2.tai + TimeDelta(365., format='jd', scale=None)
<Time object: scale='tai' format='iso' value=2011-12-31 23:59:27.068>
```

Note

Since internally **Time** uses floating point numbers, round-off errors can cause two times to be not strictly equal even if mathematically they should be. For times in UTC in particular, this can lead to surprising behavior, because when you add a **TimeDelta**, which cannot have a scale of UTC, the UTC time is first converted to TAI, then the addition is done, and finally the time is converted back to UTC. Hence, rounding errors can be incurred, which means that even expected equalities may not hold:

```
>>> t = Time(2450000., 1e-6, format='jd')
>>> t + TimeDelta(0, format='jd') == t
False
```

Barycentric and Heliocentric Light Travel Time Corrections

The arrival times of photons at an observatory are not particularly useful for accurate timing work, such as eclipse/transit timing of binaries or exoplanets. This is because the changing location of the observatory causes photons to arrive early or late. The solution is to calculate the time the photon would have arrived at a standard location; either the Solar System barycenter or the heliocenter.

Example

Suppose you observed the dwarf nova IP Peg from Greenwich and have a list of times in MJD form, in the UTC timescale. You then create appropriate **`Time`** and **`SkyCoord`** objects and calculate light travel times to the barycenter as follows:

```
>>> from astropy import time, coordinates as coord, units as u
>>> ip_peg = coord.SkyCoord("23:23:08.55", "+18:24:59.3",
...                         unit=(u.hourangle, u.deg), frame='icrs')
>>> greenwich = coord.EarthLocation.of_site('greenwich')
>>> times = time.Time([56325.95833333, 56325.978254], format='mjd',
...                   scale='utc', location=greenwich)
>>> ltt_bary = times.light_travel_time(ip_peg)
>>> ltt_bary
<TimeDelta object: scale='tdb' format='jd' value=[-0.0037715
-0.00377286]>
```

If you desire the light travel time to the heliocenter instead, then use:

```
>>> ltt_helio = times.light_travel_time(ip_peg, 'heliocentric')
>>> ltt_helio
<TimeDelta object: scale='tdb' format='jd' value=[-0.00376576
-0.00376712]>
```

The method returns an **`TimeDelta`** object, which can be added to your times to give the arrival time of the photons at the barycenter or heliocenter. Here, you should be careful with the timescales used; for more detailed information about timescales, see Time Scale.

The heliocenter is not a fixed point, and therefore the gravity continually changes at the heliocenter. Thus, the use of a relativistic timescale like TDB is not particularly appropriate, and, historically, times corrected to the heliocenter are given in the UTC timescale:

```
>>> times_heliocentre = times.utc + ltt_helio
```

Corrections to the barycenter are more precise than the heliocenter, because the barycenter is a fixed point where gravity is constant. For maximum accuracy you want to have your barycentric corrected times in a timescale that has always ticked at a uniform rate, and ideally one whose tick rate is related to the rate that a clock would tick at the barycenter. For this reason, barycentric corrected times normally use the TDB timescale:

```
>>> time_barycentre = times.tdb + ltt_bary
```

By default, the light travel time is calculated using the position and velocity of Earth and the Sun from ERFA routines, but you can also get more precise calculations using the JPL ephemerides (which are derived from dynamical models). An example using the JPL ephemerides is:

```
>>> ltt_bary_jpl = times.light_travel_time(ip_peg, ephemeris='jpl')
>>> ltt_bary_jpl
<TimeDelta object: scale='tdb' format='jd' value=[-0.0037715
-0.00377286]>
>>> (ltt_bary_jpl - ltt_bary).to(u.ms)
<Quantity [-0.00132325, -0.00132861] ms>
```

The difference between the built-in ephemerides and the JPL ephemerides is normally of the order of 1/100th of a millisecond, so the built-in ephemerides should be suitable for most purposes. For more details about what ephemerides are available, including the requirements for using JPL ephemerides, see Solar System Ephemerides.

Interaction with Time-Like Quantities

Where possible, **`Quantity`** objects with units of time are treated as **`TimeDelta`** objects with undefined scale (though necessarily with lower precision). They can also be used as input in constructing **`Time`** and **`TimeDelta`** objects, and **`TimeDelta`** objects can be converted to **`Quantity`** objects of arbitrary units of time.

Examples

To use **Quantity** objects with units of time:

```
>>> import astropy.units as u
>>> Time(10.*u.yr, format='gps')   # time-valued quantities can be used for
...                                # for formats requiring a time offset
<Time object: scale='tai' format='gps' value=315576000.0>
>>> Time(10.*u.yr, 1.*u.s, format='gps')
<Time object: scale='tai' format='gps' value=315576001.0>
>>> Time(2000.*u.yr, format='jyear')
<Time object: scale='tt' format='jyear' value=2000.0>
>>> Time(2000.*u.yr, format='byear')
...                                # but not for Besselian year, which implies
...                                # a different time scale
...
Traceback (most recent call last):
  ...
ValueError: Input values did not match the format class byear:
ValueError: Cannot use Quantities for 'byear' format, as the interpretation would be ambiguous. Use float with Besselian year instead.

>>> TimeDelta(10.*u.yr)            # With a quantity, no format is required
<TimeDelta object: scale='None' format='jd' value=3652.5>

>>> dt = TimeDelta([10., 20., 30.], format='jd')
>>> dt.to(u.hr)                    # can convert TimeDelta to a quantity
<Quantity [240., 480., 720.] h>
>>> dt > 400. * u.hr               # and compare to quantities with units of time
array([False,  True,  True]...)
>>> dt + 1.*u.hr                   # can also add/subtract such quantities
<TimeDelta object: scale='None' format='jd' value=[10.04166667 20.04166667 30.04166667]>
>>> Time(50000., format='mjd', scale='utc') + 1.*u.hr
<Time object: scale='utc' format='mjd' value=50000.0416667>
>>> dt * 10.*u.km/u.s              # for multiplication and division with a
...                                # Quantity, TimeDelta is converted
<Quantity [100., 200., 300.] d km / s>
>>> dt * 10.*u.Unit(1)             # unless the Quantity is
```

```
dimensionless
<TimeDelta object: scale='None' format='jd' value=[100. 200. 300.]>
```

Writing a Custom Format

Some applications may need a custom `Time` format, and this capability is available by making a new subclass of the `TimeFormat` class. When such a subclass is defined in your code, the format class and corresponding name is automatically registered in the set of available time formats.

Examples

The key elements of a new format class are illustrated by examining the code for the `jd` format (which is one of the most minimal):

```
class TimeJD(TimeFormat):
    """
    Julian Date time format.
    """
    name = 'jd'  # Unique format name

    def set_jds(self, val1, val2):
        """
        Set the internal jd1 and jd2 values from the input val1,
val2.
        The input values are expected to conform to this format, as
        validated by self._check_val_type(val1, val2) during
__init__.
        """
        self._check_scale(self._scale)  # Validate scale.
        self.jd1, self.jd2 = day_frac(val1, val2)

    @property
    def value(self):
        """
        Return format ``value`` property from internal jd1, jd2
        """
        return self.jd1 + self.jd2
```

As mentioned above, the `_check_val_type(self, val1, val2)` method may need to be overridden to validate the inputs as conforming to the format specification. By default this checks for valid float, float array, or `Quantity` inputs. In contrast, the `iso` format class ensures the inputs meet the ISO format specification for strings.

One special case that is relatively common and more convenient to implement is a format that makes a small change to the date format. For instance, you could insert `T` in the `yday` format with the following `TimeYearDayTimeCustom` class. Notice how the `subfmts` definition is modified slightly from the standard **`TimeISO`** class from which it inherits:

```
>>> from astropy.time import TimeISO
>>> class TimeYearDayTimeCustom(TimeISO):
...    """
...    Year, day-of-year and time as "<YYYY>-<DOY>T<HH>:
<MM>:<SS.sss...>".
...    The day-of-year (DOY) goes from 001 to 365 (366 in leap
years).
...    For example, 2000-001T00:00:00.000 is midnight on January 1,
2000.
...    The allowed subformats are:
...    - 'date_hms': date + hours, mins, secs (and optional
fractional secs)
...    - 'date_hm': date + hours, mins
...    - 'date': date
...    """
...    name = 'yday_custom'  # Unique format name
...    subfmts = (('date_hms',
...                '%Y-%jT%H:%M:%S',
...                '{year:d}-{yday:03d}T{hour:02d}:{min:02d}:
{sec:02d}'),
...               ('date_hm',
...                '%Y-%jT%H:%M',
...                '{year:d}-{yday:03d}T{hour:02d}:{min:02d}'),
...               ('date',
...                '%Y-%j',
...                '{year:d}-{yday:03d}'))


>>> t = Time('2000-01-01')
>>> t.yday_custom
'2000-001T00:00:00.000'
>>> t2 = Time('2016-001T00:00:00')
>>> t2.iso
'2016-01-01 00:00:00.000'
```

Another special case that is relatively common is a format that represents the time since a particular epoch. The classic example is Unix time which is the number of seconds since 1970-01-01 00:00:00 UTC, not counting leap seconds. What if we wanted that value but **do** want to count leap seconds. This would be done by using the TAI scale instead of the UTC scale. In this case we inherit from the **`TimeFromEpoch`** class and define a few class attributes:

```
>>> from astropy.time.formats import erfa, TimeFromEpoch
>>> class TimeUnixLeap(TimeFromEpoch):
...     """
...     Seconds from 1970-01-01 00:00:00 TAI.  Similar to Unix time
...     but this includes leap seconds.
...     """
...     name = 'unix_leap'
...     unit = 1.0 / erfa.DAYSEC  # in days (1 day == 86400 seconds)
...     epoch_val = '1970-01-01 00:00:00'
...     epoch_val2 = None
...     epoch_scale = 'tai'  # Scale for epoch_val class attribute
...     epoch_format = 'iso'  # Format for epoch_val class attribute

>>> t = Time('2000-01-01')
>>> t.unix_leap
946684832.0
>>> t.unix_leap - t.unix
32.0
```

Going beyond this will probably require looking at the `astropy` code for more guidance, but if you get stuck, the `astropy` developers are more than happy to help. If you write a format class that is widely useful we might want to include it in the core!

Timezones

When a **`Time`** object is constructed from a timezone-aware **`datetime`**, no timezone information is saved in the **`Time`** object. However, **`Time`** objects can be converted to timezone-aware datetime objects.

Example

To convert a **`Time`** object to a timezone-aware datetime object:

```
>>> from datetime import datetime
>>> from astropy.time import Time, TimezoneInfo
>>> import astropy.units as u
>>> utc_plus_one_hour = TimezoneInfo(utc_offset=1*u.hour)
>>> dt_aware = datetime(2000, 1, 1, 0, 0, 0,
tzinfo=utc_plus_one_hour)
>>> t = Time(dt_aware)  # Loses timezone info, converts to UTC
>>> print(t)            # will return UTC
1999-12-31 23:00:00
>>> print(t.to_datetime(timezone=utc_plus_one_hour)) # to timezone-
aware datetime
```

```
2000-01-01 00:00:00+01:00
```

Timezone database packages, like pytz for example, may be more convenient to use to create `tzinfo` objects used to specify timezones rather than the `TimezoneInfo` object.

Custom String Formats with `strftime` and `strptime`

The `Time` object supports output string representation using the format specification language defined in the Python standard library for `time.strftime`. This can be done using the `strftime` method.

Examples

To get output string representation using the `strftime` method:

```
>>> from astropy.time import Time
>>> t = Time('2018-01-01T10:12:58')
>>> t.strftime('%H:%M:%S %d %b %Y')
'10:12:58 01 Jan 2018'
```

Conversely, to create a `Time` object from a custom date string that can be parsed with Python standard library `time.strptime` (using the same format language linked above), use the `strptime` class method:

```
>>> from astropy.time import Time
>>> t = Time.strptime('23:59:60 30 June 2015', '%H:%M:%S %d %B %Y')
>>> t
<Time object: scale='utc' format='isot'
value=2015-06-30T23:59:60.000>
```

Reference/API

astropy.time Package

Functions

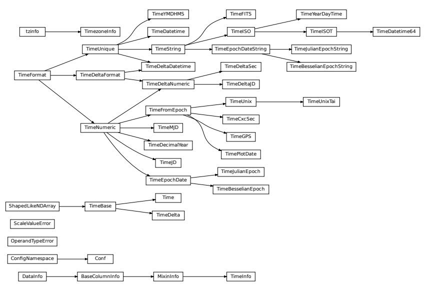
`update_leap_seconds`([files])	If the current ERFA leap second table is out of date, try to update it.

Classes

Conf()	Configuration parameters for astropy.table.
OperandTypeError(left, right[, op])	
ScaleValueError	
Time(val[, val2, format, scale, precision, …])	Represent and manipulate times and dates for astronomy.
TimeBase()	Base time class from which Time and TimeDelta inherit.
TimeBesselianEpoch(val1, val2, scale, …[, …])	Besselian Epoch year as floating point value(s) like 1950.0
TimeBesselianEpochString(val1, val2, scale, …)	Besselian Epoch year as string value(s) like ‘B1950.0’
TimeCxcSec(val1, val2, scale, precision, …)	Chandra X-ray Center seconds from 1998-01-01 00:00:00 TT.
TimeDatetime(val1, val2, scale, precision, …)	Represent date as Python standard library datetime object
TimeDatetime64(val1, val2, scale, precision, …)	
TimeDecimalYear(val1, val2, scale, …[, …])	Time as a decimal year, with integer values corresponding to midnight of the first day of each year.
TimeDelta(val[, val2, format, scale, …])	Represent the time difference between two times.
TimeDeltaDatetime(val1, val2, scale, …[, …])	Time delta in datetime.timedelta
TimeDeltaFormat(val1, val2, scale, …[, …])	Base class for time delta representations
TimeDeltaJD(val1, val2, scale, precision, …)	Time delta in Julian days (86400 SI seconds)
TimeDeltaNumeric(val1, val2, scale, …[, …])	
TimeDeltaSec(val1, val2, scale, precision, …)	Time delta in SI seconds
TimeEpochDate(val1, val2, scale, precision, …)	Base class for support floating point Besselian and Julian epoch dates
TimeEpochDateString(val1, val2, scale, …)	Base class to support string Besselian and Julian epoch dates such as ‘B1950.0’ or ‘J2000.0’ respectively.
TimeFITS(val1, val2, scale, precision, …)	FITS format: “[±Y]YYYY-MM-DD[THH:MM:SS[.sss]]”.
TimeFormat(val1, val2, scale, precision, …)	Base class for time representations.
TimeFromEpoch(val1, val2, scale, precision, …)	Base class for times that represent the interval from a particular epoch as a floating point multiple of a unit time interval (e.g.
TimeGPS(val1, val2, scale, precision, …[, …])	GPS time: seconds from 1980-01-06 00:00:00 UTC For example, 630720013.0 is midnight on January 1, 2000.
TimeISO(val1, val2, scale, precision, …[, …])	ISO 8601 compliant date-time format “YYYY-MM-DD HH:MM:SS.sss…”.
TimeISOT(val1, val2, scale, precision, …)	ISO 8601 compliant date-time format “YYYY-MM-DDTHH:MM:SS.sss…”.
TimeInfo([bound])	Container for meta information like name, description, format.
TimeJD(val1, val2, scale, precision, …[, …])	Julian Date time format.

TimeJulianEpoch(val1, val2, scale, …[, …])	Julian Epoch year as floating point value(s) like 2000.0
TimeJulianEpochString(val1, val2, scale, …)	Julian Epoch year as string value(s) like ‘J2000.0’
TimeMJD(val1, val2, scale, precision, …[, …])	Modified Julian Date time format.
TimeNumeric(val1, val2, scale, precision, …)	
TimePlotDate(val1, val2, scale, precision, …)	Matplotlib **plot_date** input: 1 + number of days from 0001-01-01 00:00:00 UTC
TimeString(val1, val2, scale, precision, …)	Base class for string-like time representations.
TimeUnique(val1, val2, scale, precision, …)	Base class for time formats that can uniquely create a time object without requiring an explicit format specifier.
TimeUnix(val1, val2, scale, precision, …)	Unix time: seconds from 1970-01-01 00:00:00 UTC, ignoring leap seconds.
TimeUnixTai(val1, val2, scale, precision, …)	Seconds from 1970-01-01 00:00:08 TAI (see notes), including leap seconds.
TimeYMDHMS(val1, val2, scale, precision, …)	ymdhms: A Time format to represent Time as year, month, day, hour, minute, second (thus the name ymdhms).
TimeYearDayTime(val1, val2, scale, …[, …])	Year, day-of-year and time as “YYYY:DOY:HH:MM:SS.sss…”.
TimezoneInfo([utc_offset, dst, tzname])	Subclass of the **tzinfo** object, used in the to_datetime method to specify timezones.

Class Inheritance Diagram



Acknowledgments and Licenses

This package makes use of the PyERFA wrappers of the ERFA ANSI C library. The copyright of the ERFA software belongs to the NumFOCUS Foundation. The library is made available under the terms of the “BSD-three clauses” license.

The ERFA library is derived, with permission, from the International Astronomical Union’s “Standards of Fundamental Astronomy” (SOFA) library, available from http://www.iausofa.org.

Time Series (`astropy.timeseries`)

Introduction

From sampling a continuous variable at fixed times to counting events binned into time windows, many different areas of astrophysics require the manipulation of 1D time series data. To address this need, the **`astropy.timeseries`** subpackage provides classes to represent and manipulate time series.

The time series classes presented below are **`QTable`** subclasses that have special columns to represent times using the **`Time`** class. Therefore, much of the functionality described in Data Tables (astropy.table) applies here. But the main purpose of the new classes are to provide time series-specific functionality above and beyond **`QTable`**.

Getting Started

In this section, we take a quick look at how to read in a time series, access the data, and carry out some basic analysis. For more details about creating and using time series, see the full documentation in Using timeseries.

The most basic time series class is **`TimeSeries`** — it represents a time series as a collection of values at specific points in time. If you are interested in representing time series as measurements in discrete time bins, you will likely be interested in the **`BinnedTimeSeries`** subclass which we show in Using timeseries).

To start off, we retrieve a FITS file containing a Kepler light curve for a source:

```
>>> from astropy.utils.data import get_pkg_data_filename
>>> filename =
get_pkg_data_filename('timeseries/kplr010666592-2009131110544_slc.fit
s')
```

Note

The light curve provided here is handpicked for example purposes. For more information about the Kepler FITS format, see the Kepler Data Validation Document and the Kepler Science Center Light Curve Files documentation. To get other Kepler light curves for science purposes using Python, see the astroquery affiliated package.

We can then use the **`TimeSeries`** class to read in this file:

```
>>> from astropy.timeseries import TimeSeries
>>> ts = TimeSeries.read(filename, format='kepler.fits')
```

Time series are specialized kinds of **Table** objects:

```
>>> ts
<TimeSeries length=14280>
          time                timecorr    ...   pos_corr1      pos_corr2
                                 d        ...      pix            pix
         object               float32     ...    float32        float32
----------------------- ------------- ... --------------
--------------
2009-05-02T00:41:40.338  6.630610e-04 ...  1.5822421e-03 -1.4463664e-
03
2009-05-02T00:42:39.188  6.630857e-04 ...  1.5743829e-03 -1.4540013e-
03
2009-05-02T00:43:38.045  6.631103e-04 ...  1.5665225e-03 -1.4616371e-
03
2009-05-02T00:44:36.894  6.631350e-04 ...  1.5586632e-03 -1.4692718e-
03
2009-05-02T00:45:35.752  6.631597e-04 ...  1.5508028e-03 -1.4769078e-
03
2009-05-02T00:46:34.601  6.631844e-04 ...  1.5429436e-03 -1.4845425e-
03
2009-05-02T00:47:33.451  6.632091e-04 ...  1.5350844e-03 -1.4921773e-
03
2009-05-02T00:48:32.291  6.632337e-04 ...  1.5272264e-03 -1.4998110e-
03
2009-05-02T00:49:31.149  6.632584e-04 ...  1.5193661e-03
-1.5074468e-03
                    ...           ... ...            ...
...
2009-05-11T17:58:22.526  1.014493e-03 ...  3.6121816e-03  3.1950327e-
03
2009-05-11T17:59:21.376  1.014518e-03 ...  3.6102540e-03  3.1872767e-
03
2009-05-11T18:00:20.225  1.014542e-03 ...  3.6083264e-03  3.1795206e-
03
2009-05-11T18:01:19.065  1.014567e-03 ...  3.6063993e-03  3.1717657e-
03
2009-05-11T18:02:17.923  1.014591e-03 ...  3.6044715e-03  3.1640085e-
03
2009-05-11T18:03:16.772  1.014615e-03 ...  3.6025438e-03  3.1562524e-
03
2009-05-11T18:04:15.630  1.014640e-03 ...  3.6006160e-03  3.1484952e-
03
2009-05-11T18:05:14.479  1.014664e-03 ...  3.5986886e-03  3.1407392e-
03
```

```
2009-05-11T18:06:13.328  1.014689e-03 ...  3.5967610e-03  3.1329831e-
03
2009-05-11T18:07:12.186  1.014713e-03 ...  3.5948332e-03
3.1252259e-03
```

In the same way as for **`Table`** objects, the various columns and rows of **`TimeSeries`** objects can be accessed and sliced using index notation:

```
>>> ts['sap_flux']
<Quantity [1027045.06, 1027184.44, 1027076.25, ..., 1025451.56,
1025468.5 ,
           1025930.9 ] electron / s>

>>> ts['time', 'sap_flux']
<TimeSeries length=14280>
          time             sap_flux
                         electron / s
         object            float32
----------------------- --------------
2009-05-02T00:41:40.338  1.0270451e+06
2009-05-02T00:42:39.188  1.0271844e+06
2009-05-02T00:43:38.045  1.0270762e+06
2009-05-02T00:44:36.894  1.0271414e+06
2009-05-02T00:45:35.752  1.0271569e+06
2009-05-02T00:46:34.601  1.0272296e+06
2009-05-02T00:47:33.451  1.0273199e+06
2009-05-02T00:48:32.291  1.0271497e+06
2009-05-02T00:49:31.149  1.0271755e+06
                    ...            ...
2009-05-11T17:58:22.526  1.0234769e+06
2009-05-11T17:59:21.376  1.0234574e+06
2009-05-11T18:00:20.225  1.0238128e+06
2009-05-11T18:01:19.065  1.0243234e+06
2009-05-11T18:02:17.923  1.0244257e+06
2009-05-11T18:03:16.772  1.0248654e+06
2009-05-11T18:04:15.630  1.0250156e+06
2009-05-11T18:05:14.479  1.0254516e+06
2009-05-11T18:06:13.328  1.0254685e+06
2009-05-11T18:07:12.186  1.0259309e+06

>>> ts[0:4]
<TimeSeries length=4>
          time             timecorr    ...   pos_corr1      pos_corr2
                              d        ...      pix            pix
         object            float32     ...    float32        float32
----------------------- ------------- ... --------------
--------------
2009-05-02T00:41:40.338  6.630610e-04 ...  1.5822421e-03 -1.4463664e-
```

```
03
2009-05-02T00:42:39.188  6.630857e-04 ...  1.5743829e-03 -1.4540013e-
03
2009-05-02T00:43:38.045  6.631103e-04 ...  1.5665225e-03 -1.4616371e-
03
2009-05-02T00:44:36.894  6.631350e-04 ...  1.5586632e-03
-1.4692718e-03
```

As seen in the previous examples, **`TimeSeries`** objects have a `time` column, which is always the first column. This column can also be accessed using the `.time` attribute:

```
>>> ts.time
<Time object: scale='tdb' format='isot' value=
['2009-05-02T00:41:40.338' '2009-05-02T00:42:39.188'
  '2009-05-02T00:43:38.045' ... '2009-05-11T18:05:14.479'
  '2009-05-11T18:06:13.328' '2009-05-11T18:07:12.186']>
```

The first column is always a **`Time`** object (see Times and Dates), which therefore supports the ability to convert to different time scales and formats:

```
>>> ts.time.mjd
array([54953.0289391 , 54953.02962023, 54953.03030145, ...,
       54962.7536398 , 54962.75432093, 54962.75500215])

>>> ts.time.unix
array([1.24122483e+09, 1.24122489e+09, 1.24122495e+09, ...,
       1.24206505e+09, 1.24206511e+09, 1.24206517e+09])
```

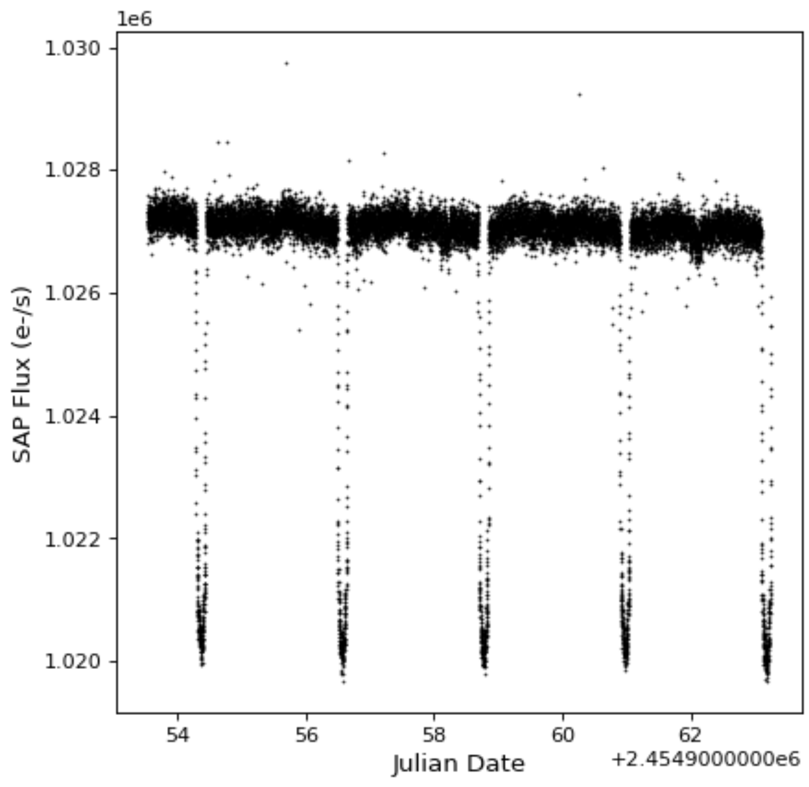
We can also check what time scale the time is defined on:

```
>>> ts.time.scale
'tdb'
```

This is the Barycentric Dynamical Time scale (see Time and Dates (astropy.time) for more details). We can use what we have seen so far to make a plot:

```
import matplotlib.pyplot as plt
plt.plot(ts.time.jd, ts['sap_flux'], 'k.', markersize=1)
plt.xlabel('Julian Date')
plt.ylabel('SAP Flux (e-/s)')
```

(png, svg, pdf)



It looks like there are a few transits! We can use the **BoxLeastSquares** class to estimate the period, using the “box least squares” (BLS) algorithm:

```
>>> import numpy as np
>>> from astropy import units as u
>>> from astropy.timeseries import BoxLeastSquares
>>> periodogram = BoxLeastSquares.from_timeseries(ts, 'sap_flux')
```

To run the periodogram analysis, we use a box with a duration of 0.2 days:

```
>>> results = periodogram.autopower(0.2 * u.day)
>>> best = np.argmax(results.power)
>>> period = results.period[best]
>>> period
<Quantity 2.20551724 d>
>>> transit_time = results.transit_time[best]
>>> transit_time
<Time object: scale='tdb' format='isot'
value=2009-05-02T20:51:16.338>
```

For more information on available periodogram algorithms, see Periodogram Algorithms.

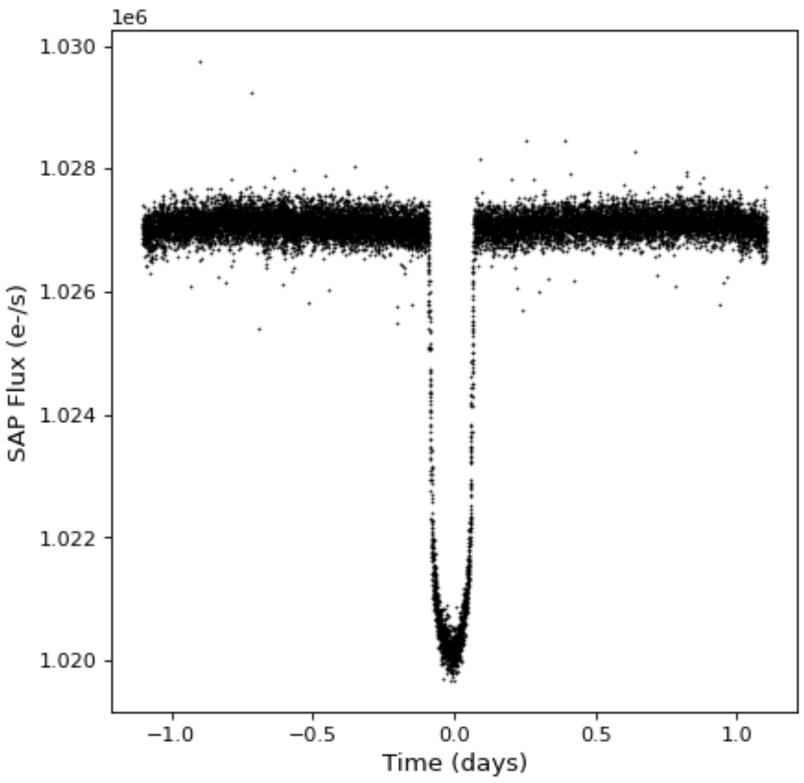
We can now fold the time series using the period we found above using the **fold()** method:

```
>>> ts_folded = ts.fold(period=period, epoch_time=transit_time)
```

Now we can take a look at the folded time series:

```
plt.plot(ts_folded.time.jd, ts_folded['sap_flux'], 'k.',
markersize=1)
plt.xlabel('Time (days)')
plt.ylabel('SAP Flux (e-/s)')
```

(png, svg, pdf)



Using the Astrostatistics Tools (astropy.stats) module, we can normalize the flux by sigma-clipping the data to determine the baseline flux:

```
>>> from astropy.stats import sigma_clipped_stats
>>> mean, median, stddev = sigma_clipped_stats(ts_folded['sap_flux'])
>>> ts_folded['sap_flux_norm'] = ts_folded['sap_flux'] / median
```

And we can downsample the time series by binning the points into bins of equal time — this returns a **BinnedTimeSeries**:

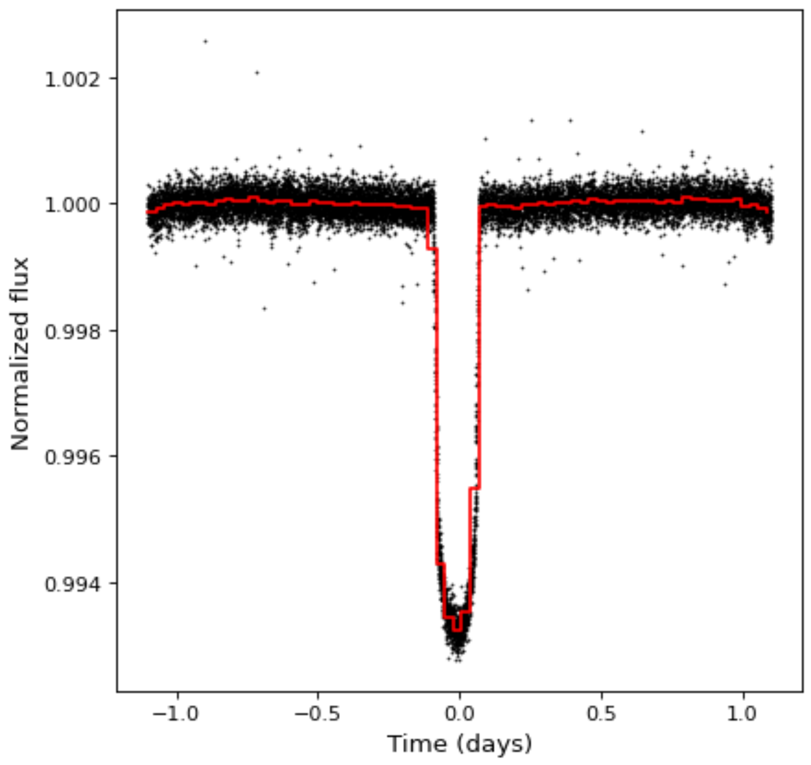
```
>>> from astropy.timeseries import aggregate_downsample
>>> ts_binned = aggregate_downsample(ts_folded, time_bin_size=0.03 *
u.day)
>>> ts_binned
```

```
<BinnedTimeSeries length=74>
   time_bin_start     time_bin_size    ...   sap_flux_norm
                            s          ...
       object            float64       ...        float64
------------------- ------------------ ... ------------------
-1.1022116370482966             2592.0 ... 0.9998741745948792
-1.0722116370482966             2592.0 ... 0.9999074339866638
-1.0422116370482966             2592.0 ...  0.999972939491272
-1.0122116370482965             2592.0 ... 1.0000077486038208
-0.9822116370482965             2592.0 ... 0.9999921917915344
-0.9522116370482965             2592.0 ... 1.0000101327896118
-0.9222116370482966             2592.0 ... 1.0000121593475342
-0.8922116370482965             2592.0 ... 0.9999905228614807
-0.8622116370482965 2592.0000000000023 ... 1.0000263452529907
                ...                ... ...                ...
 0.8177883629517035 2591.9999999999977 ... 1.0000624656677246
 0.8477883629517035 2592.0000000000014 ... 1.0000633001327515
 0.8777883629517035  2592.000000000019 ... 1.0000433921813965
 0.9077883629517037 2591.9999999999814 ...  1.000024676322937
 0.9377883629517034   2592.00000000002 ... 1.0000224113464355
 0.9677883629517037  2591.999999999981 ... 1.0000698566436768
 0.9977883629517035             2592.0 ... 0.9999606013298035
 1.0277883629517035             2592.0 ... 0.9999635815620422
 1.0577883629517035             2592.0 ... 0.9999105930328369
 1.0877883629517036 2592.0000000000095 ... 0.9998687505722046
```

Now we can take a look at the final result:

```
plt.plot(ts_folded.time.jd, ts_folded['sap_flux_norm'], 'k.',
markersize=1)
plt.plot(ts_binned.time_bin_start.jd, ts_binned['sap_flux_norm'],
'r-', drawstyle='steps-post')
plt.xlabel('Time (days)')
plt.ylabel('Normalized flux')
```

(png, svg, pdf)



To learn more about the capabilities in the **`astropy.timeseries`** module, you can find links to the full documentation in the next section.

Using `timeseries`

The details of using **`astropy.timeseries`** are provided in the following sections:

Initializing and Reading in Time Series

Creating Time Series

Initializing a Time Series

The first type of time series that we will look at here is a **`TimeSeries`** object, which can be used for a time series which samples a continuous variable at discrete, instantaneous times. Initializing a **`TimeSeries`** object can be done in the same ways as initializing a **`Table`** object (see Data Tables), but additional arguments related to the times should be specified.

Evenly Sampled Time Series

The most convenient way to construct an evenly sampled **`TimeSeries`** is to specify the start time, the time interval, and the number of samples:

```
>>> from astropy import units as u
>>> from astropy.timeseries import TimeSeries
>>> ts1 = TimeSeries(time_start='2016-03-22T12:30:31',
...                  time_delta=3 * u.s,
...                  n_samples=5)
>>> ts1
<TimeSeries length=5>
          time
         object
-----------------------
2016-03-22T12:30:31.000
2016-03-22T12:30:34.000
2016-03-22T12:30:37.000
2016-03-22T12:30:40.000
2016-03-22T12:30:43.000
```

The `time` keyword argument can be set to anything that can be passed to the **`Time`** class (see also Time and Dates) or **`Time`** objects directly. Note that the `n_samples` argument is only needed if you are not also passing in data during initialization (see Passing Data During Initialization).

Arbitrarily Sampled Time Series

To construct a sampled time series with samples at arbitrary times, you can pass multiple times to the `time` argument:

```
>>> ts2 = TimeSeries(time=['2016-03-22T12:30:31',
...                        '2016-03-22T12:30:38',
...                        '2016-03-22T12:34:40'])
>>> ts2
<TimeSeries length=3>
          time
         object
-----------------------
2016-03-22T12:30:31.000
2016-03-22T12:30:38.000
2016-03-22T12:34:40.000
```

You can also specify a vector **`Time`** object directly as the `time=` argument, or a vector **`TimeDelta`** argument or a quantity array to the `time_delta=` argument.:

```
>>> TimeSeries(time_start="2011-01-01T00:00:00",
...            time_delta=[0.1, 0.2, 0.1, 0.3, 0.2]*u.s)
<TimeSeries length=5>
          time
        object
```

```
-----------------------
2011-01-01T00:00:00.000
2011-01-01T00:00:00.100
2011-01-01T00:00:00.300
2011-01-01T00:00:00.400
2011-01-01T00:00:00.700
```

Initializing a Binned Time Series

The `BinnedTimeSeries` can be used to represent time series where each entry corresponds to measurements taken over a range in time — for instance, a light curve constructed by binning X-ray photon events. This class supports equal-size or uneven bins, and contiguous and non-contiguous bins. As for `TimeSeries`, initializing a `BinnedTimeSeries` can be done in the same ways as initializing a `Table` object (see Data Tables), but additional arguments related to the times should be specified as described below.

Equal-Sized Contiguous Bins

To create a binned time series with equal-size contiguous bins, it is sufficient to specify a start time as well as a bin size:

```
>>> from astropy.timeseries import BinnedTimeSeries
>>> ts3 = BinnedTimeSeries(time_bin_start='2016-03-22T12:30:31',
...                        time_bin_size=3 * u.s, n_bins=10)
>>> ts3
<BinnedTimeSeries length=10>
    time_bin_start     time_bin_size
                             s
        object            float64
----------------------- -------------
2016-03-22T12:30:31.000           3.0
2016-03-22T12:30:34.000           3.0
2016-03-22T12:30:37.000           3.0
2016-03-22T12:30:40.000           3.0
2016-03-22T12:30:43.000           3.0
2016-03-22T12:30:46.000           3.0
2016-03-22T12:30:49.000           3.0
2016-03-22T12:30:52.000           3.0
2016-03-22T12:30:55.000           3.0
2016-03-22T12:30:58.000           3.0
```

Note that the `n_bins` argument is only needed if you are not also passing in data during initialization (see Passing Data During Initialization).

Uneven Contiguous Bins

When creating a binned time series with uneven contiguous bins, the bin size can be changed to give multiple values (note that in this case `n_bins` is not required):

```
>>> ts4 = BinnedTimeSeries(time_bin_start='2016-03-22T12:30:31',
...                        time_bin_size=[3, 3, 2, 3] * u.s)
>>> ts4
<BinnedTimeSeries length=4>
     time_bin_start     time_bin_size
                              s
         object            float64
----------------------- -------------
2016-03-22T12:30:31.000           3.0
2016-03-22T12:30:34.000           3.0
2016-03-22T12:30:37.000           2.0
2016-03-22T12:30:39.000           3.0
```

Alternatively, you can create the same time series by giving an array of start times as well as a single end time:

```
>>> ts5 = BinnedTimeSeries(time_bin_start=['2016-03-22T12:30:31',
...                                         '2016-03-22T12:30:34',
...                                         '2016-03-22T12:30:37',
...                                         '2016-03-22T12:30:39'],
...                        time_bin_end='2016-03-22T12:30:42')
>>> ts5
<BinnedTimeSeries length=4>
    time_bin_start          time_bin_size
                                  s
      object                   float64
----------------------- -----------------
2016-03-22T12:30:31.000               3.0
2016-03-22T12:30:34.000               3.0
2016-03-22T12:30:37.000               2.0
2016-03-22T12:30:39.000               3.0
```

Uneven Non-Contiguous Bins

To create a binned time series with non-contiguous bins, you can either specify an array of start times and bin widths:

```
>>> ts6 = BinnedTimeSeries(time_bin_start=['2016-03-22T12:30:31',
...                                         '2016-03-22T12:30:38',
...                                         '2016-03-22T12:34:40'],
...                        time_bin_size=[5, 100, 2]*u.s)
>>> ts6
<BinnedTimeSeries length=3>
     time_bin_start     time_bin_size
                              s
         object            float64
----------------------- -------------
2016-03-22T12:30:31.000           5.0
```

```
2016-03-22T12:30:38.000         100.0
2016-03-22T12:34:40.000           2.0
```

Or in the most general case, you can also specify multiple times for `time_bin_start` and `time_bin_end`:

```
>>> ts7 = BinnedTimeSeries(time_bin_start=['2016-03-22T12:30:31',
...                                        '2016-03-22T12:30:33',
...                                        '2016-03-22T12:30:40'],
...                        time_bin_end=['2016-03-22T12:30:32',
...                                      '2016-03-22T12:30:35',
...                                      '2016-03-22T12:30:41'])
>>> ts7
<BinnedTimeSeries length=3>
    time_bin_start       time_bin_size
                               s
         object             float64
----------------------- -----------------
2016-03-22T12:30:31.000               1.0
2016-03-22T12:30:33.000               2.0
2016-03-22T12:30:40.000               1.0
```

Adding Data to the Time Series

The above examples show how to initialize **`TimeSeries`** objects, but these do not include any data aside from the times. There are different ways of adding data, as with the **`Table`** class.

Passing Data During Initialization

It is possible to pass data during the initialization of a **`TimeSeries`** object, as for **`Table`** objects. For instance:

```
>>> ts8 = BinnedTimeSeries(time_bin_start=['2016-03-22T12:30:31',
...                                        '2016-03-22T12:30:34',
...                                        '2016-03-22T12:30:37',
...                                        '2016-03-22T12:30:39'],
...                        time_bin_end='2016-03-22T12:30:42',
...                        data={'flux': [1., 4., 5., 6.] * u.mJy})
>>> ts8
<BinnedTimeSeries length=4>
       time_bin_start             time_bin_size      flux
                                        s             mJy
          object                     float64       float64
----------------------- ---------------- -------
2016-03-22T12:30:31.000              3.0     1.0
2016-03-22T12:30:34.000              3.0     4.0
2016-03-22T12:30:37.000              2.0     5.0
2016-03-22T12:30:39.000              3.0     6.0
```

Adding Data After Initialization

Once a **TimeSeries** object is initialized, you can add columns/fields to it as you would for a **Table** object:

```
>>> from astropy import units as u
>>> ts1['flux'] = [1., 4., 5., 6., 4.] * u.mJy
>>> ts1
<TimeSeries length=5>
          time            flux
                          mJy
         object         float64
----------------------- -------
2016-03-22T12:30:31.000     1.0
2016-03-22T12:30:34.000     4.0
2016-03-22T12:30:37.000     5.0
2016-03-22T12:30:40.000     6.0
2016-03-22T12:30:43.000     4.0
```

Adding Rows

Adding rows to **TimeSeries** or **BinnedTimeSeries** can be done using the **add_row()** method, as for **Table** and **Table**. This method takes a dictionary where the keys are column names:

```
>>> ts8.add_row({'time_bin_start': '2016-03-22T12:30:44.000',
...              'time_bin_size': 2 * u.s,
...              'flux': 3 * u.mJy})
>>> ts8
<BinnedTimeSeries length=5>
    time_bin_start       time_bin_size      flux
```

```
                                  s            mJy
         object                float64       float64
----------------------- ----------------- -------
2016-03-22T12:30:31.000               3.0     1.0
2016-03-22T12:30:34.000               3.0     4.0
2016-03-22T12:30:37.000               2.0     5.0
2016-03-22T12:30:39.000               3.0     6.0
2016-03-22T12:30:44.000               2.0     3.0
```

If you want to be able to skip some values when adding rows, you should make sure that masking is enabled — see Masking Values in Time Series for more details.

Reading and Writing Time Series

Built-in Readers

Since **`TimeSeries`** and **`BinnedTimeSeries`** are subclasses of **`Table`**, they have **`read()`** and **`write()`** methods that can be used to read and write time series from files. We include a few readers for well-defined formats in **`astropy.timeseries`**. For instance we have readers for light curves in FITS format from the Kepler and TESS missions.

Example

In this demonstration of using Kepler FITS time series, we start off by fetching an example file:

```
from astropy.utils.data import get_pkg_data_filename
example_data =
get_pkg_data_filename('timeseries/kplr010666592-2009131110544_slc.fit
s')
```

Note

The light curve provided here is handpicked for example purposes. To get other Kepler light curves for science purposes using Python, see the astroquery affiliated package.

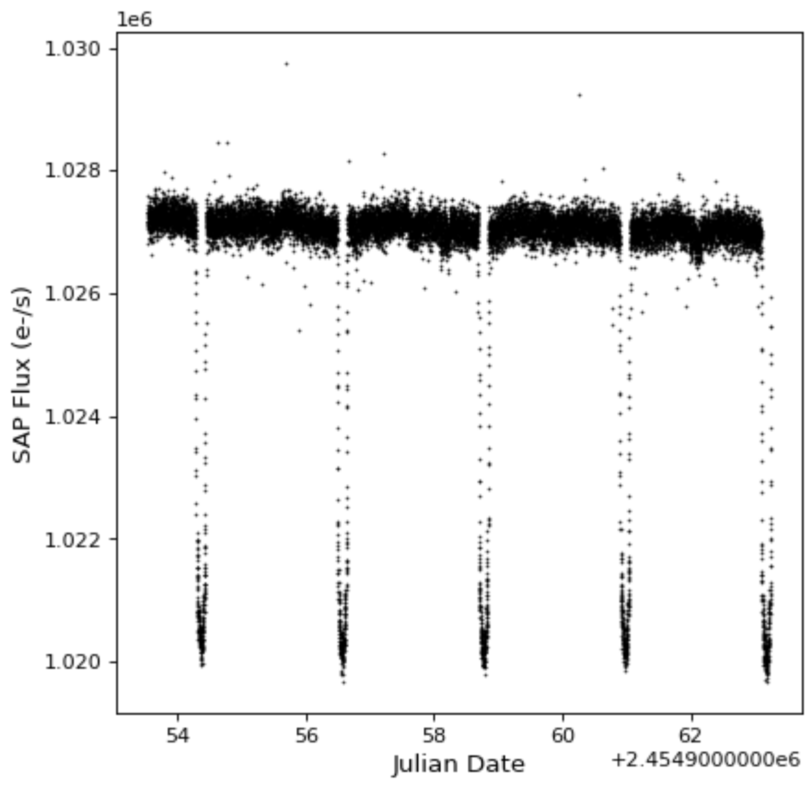
This will set `example_data` to the filename of the downloaded file (so you can replace this by the filename for the file you want to read in). We can then read in the time series using:

```
from astropy.timeseries import TimeSeries
kepler = TimeSeries.read(example_data, format='kepler.fits')
```

Now we can check that the time series has been read in correctly:

```
import matplotlib.pyplot as plt

plt.plot(kepler.time.jd, kepler['sap_flux'], 'k.', markersize=1)
plt.xlabel('Julian Date')
plt.ylabel('SAP Flux (e-/s)')
```

(png, svg, pdf)



Reading Common Light Curve Formats

At the moment only a few formats are defined in `astropy` itself, in part because there are not many well-documented formats for storing time series. So in many cases, you will likely have to first read in your files using the more generic **`Table`** class (see Reading and Writing Table Objects). In fact, the **`TimeSeries.read`** and **`BinnedTimeSeries.read`** methods can do this behind the scenes. If the table cannot be read by any of the time series readers, these methods will try to use some of the default **`Table`** readers and then require users to specify the names of the important columns.

Examples

If you are reading in a file called **`sampled.csv`** where the time column is called `Date` and is an ISO string, you can do:

```
>>> from astropy.timeseries import TimeSeries
>>> from astropy.utils.data import get_pkg_data_filename
>>> sampled_filename = get_pkg_data_filename('data/sampled.csv',
...
package='astropy.timeseries.tests')
>>> ts = TimeSeries.read(sampled_filename, format='ascii.csv',
...                      time_column='Date')
>>> ts[:3]
<TimeSeries length=3>
          time             A       B       C       D       E       F
G
         object         float64 float64 float64 float64 float64
float64 float64
----------------------- ------- ------- ------- ------- -------
------- -------
2008-03-18 00:00:00.000   24.68  164.93  114.73   26.27   19.21
28.87   63.44
2008-03-19 00:00:00.000   24.18  164.89  114.75   26.22   19.07
27.76   59.98
2008-03-20 00:00:00.000   23.99  164.63  115.04   25.78   19.01
27.04   59.61
```

If you are reading in a binned time series from a file called **binned.csv** and with a column `time_start` giving the start time and `bin_size` giving the size of each bin, you can do:

```
>>> from astropy import units as u
>>> from astropy.timeseries import BinnedTimeSeries
>>> binned_filename = get_pkg_data_filename('data/binned.csv',
...
package='astropy.timeseries.tests')
>>> ts = BinnedTimeSeries.read(binned_filename, format='ascii.csv',
...                            time_bin_start_column='time_start',
...                            time_bin_size_column='bin_size',
...                            time_bin_size_unit=u.s)
>>> ts[:3]
<BinnedTimeSeries length=3>
     time_bin_start     time_bin_size ...    E       F
                              s       ...
         object            float64    ... float64 float64
----------------------- ------------- ... ------- -------
2016-03-22T12:30:31.000           3.0 ...   28.87   63.44
2016-03-22T12:30:34.000           3.0 ...   27.76   59.98
2016-03-22T12:30:37.000           3.0 ...   27.04   59.61
```

See the documentation for **TimeSeries.read** and **BinnedTimeSeries.read** for more details.

Alternatively, you can read in the table using your own code then construct the

TimeSeries object as described in Creating Time Series, although then you cannot write out another time series in the same format.

If you have written a reader/writer for a commonly used format, please feel free to contribute it to `astropy`!

Accessing Data and Manipulating Time Series

Accessing Data in Time Series

Accessing Data

For the examples in this page, we will consider a sampled time series with two data columns — `flux` and `temp`:

```
>>> from astropy import units as u
>>> from astropy.timeseries import TimeSeries
>>> ts = TimeSeries(time_start='2016-03-22T12:30:31',
...                 time_delta=3 * u.s,
...                 data={'flux': [1., 4., 5., 3., 2.] * u.Jy,
...                       'temp': [40., 41., 39., 24., 20.] * u.K},
...                 names=('flux', 'temp'))
```

As for **Table**, columns can be accessed by name:

```
>>> ts['flux']
<Quantity [ 1., 4., 5., 3., 2.] Jy>
>>> ts['time']
<Time object: scale='utc' format='isot' value=
['2016-03-22T12:30:31.000' '2016-03-22T12:30:34.000'
 '2016-03-22T12:30:37.000' '2016-03-22T12:30:40.000'
 '2016-03-22T12:30:43.000']>
```

And rows can be accessed by index:

```
>>> ts[0]
<Row index=0>
          time              flux    temp
                             Jy      K
         object            float64 float64
----------------------- ------- -------
2016-03-22T12:30:31.000     1.0    40.0
```

Accessing individual values can then be done either by accessing a column and then a row, or vice versa:

```
>>> ts[0]['flux']
<Quantity 1. Jy>

>>> ts['temp'][2]
<Quantity 39. K>
```

Accessing Times

For `TimeSeries`, the `time` column can be accessed using the regular column access notation, as shown in Accessing Data, but it can also be accessed more conveniently using the `time` attribute:

```
>>> ts.time
<Time object: scale='utc' format='isot' value=
['2016-03-22T12:30:31.000' '2016-03-22T12:30:34.000'
 '2016-03-22T12:30:37.000' '2016-03-22T12:30:40.000'
 '2016-03-22T12:30:43.000']>
```

For `BinnedTimeSeries`, we provide three attributes: `time_bin_start`, `time_bin_center`, and `time_bin_end`:

```
>>> from astropy.timeseries import BinnedTimeSeries
>>> bts = BinnedTimeSeries(time_bin_start='2016-03-22T12:30:31',
...                        time_bin_size=3 * u.s, n_bins=5)
>>> bts.time_bin_start
<Time object: scale='utc' format='isot' value=
['2016-03-22T12:30:31.000' '2016-03-22T12:30:34.000'
 '2016-03-22T12:30:37.000' '2016-03-22T12:30:40.000'
 '2016-03-22T12:30:43.000']>
>>> bts.time_bin_center
<Time object: scale='utc' format='isot' value=
['2016-03-22T12:30:32.500' '2016-03-22T12:30:35.500'
 '2016-03-22T12:30:38.500' '2016-03-22T12:30:41.500'
 '2016-03-22T12:30:44.500']>
>>> bts.time_bin_end
<Time object: scale='utc' format='isot' value=
['2016-03-22T12:30:34.000' '2016-03-22T12:30:37.000'
 '2016-03-22T12:30:40.000' '2016-03-22T12:30:43.000'
 '2016-03-22T12:30:46.000']>
```

In addition, the `time_bin_size` attribute can be used to access the bin sizes:

```
>>> bts.time_bin_size
<Quantity [3., 3., 3., 3., 3.] s>
```

Note that only `time_bin_start` and `time_bin_size` are available as actual columns, and **time_bin_center** and **time_bin_end** are computed on the fly.

See Converting between Different Time Representations for more information about changing between different representations of time.

Extracting a Subset of Columns

We can create a new time series with just the `flux` column by doing:

```
>>> ts['time', 'flux']
<TimeSeries length=5>
          time             flux
                            Jy
         object          float64
----------------------- -------
2016-03-22T12:30:31.000     1.0
2016-03-22T12:30:34.000     4.0
2016-03-22T12:30:37.000     5.0
2016-03-22T12:30:40.000     3.0
2016-03-22T12:30:43.000     2.0
```

Note that the new columns will be copies (not views) of the original columns. We can also create a plain **QTable** by extracting just the `flux` and `temp` columns:

```
>>> ts['flux', 'temp']
<QTable length=5>
  flux    temp
   Jy      K
float64 float64
------- -------
    1.0    40.0
    4.0    41.0
    5.0    39.0
    3.0    24.0
    2.0    20.0
```

Extracting a Subset of Rows

TimeSeries objects can be sliced by rows, using the same syntax as for **Time**, for example:

```
>>> ts[0:2]
<TimeSeries length=2>
          time             flux    temp
                            Jy      K
         object          float64 float64
```

```
----------------------- ------- -------
2016-03-22T12:30:31.000     1.0    40.0
2016-03-22T12:30:34.000     4.0    41.0
```

`TimeSeries` objects are also automatically indexed using the functionality described in Table Indexing. This provides the ability to access rows and a subset of rows using the `loc` and `iloc` attributes.

The `loc` attribute can be used to slice `TimeSeries` objects by time. For example, the following can be used to extract all entries for a given timestamp:

```
>>> from astropy.time import Time
>>> ts.loc[Time('2016-03-22T12:30:31.000')]
<Row index=0>
          time             flux    temp
                            Jy      K
         object          float64 float64
----------------------- ------- -------
2016-03-22T12:30:31.000     1.0    40.0
```

Or within a time range:

```
>>> ts.loc['2016-03-22T12:30:30':'2016-03-22T12:30:41']
<TimeSeries length=4>
          time             flux    temp
                            Jy      K
         object          float64 float64
----------------------- ------- -------
2016-03-22T12:30:31.000     1.0    40.0
2016-03-22T12:30:34.000     4.0    41.0
2016-03-22T12:30:37.000     5.0    39.0
2016-03-22T12:30:40.000     3.0    24.0
```

Note that in this case we did not specify `Time` — this is not needed if the string is an ISO 8601 time string. As for the `QTable` and `Table` class `loc` attribute, in order to be consistent with pandas, the last item in the `loc` range is inclusive.

Also note that the result will always be sorted by time. Similarly, the `iloc` attribute can be used to fetch rows from the time series *sorted by time*, so for example, the first two entries (by time) can be accessed with:

```
>>> ts.iloc[0:2]
<TimeSeries length=2>
          time             flux    temp
                            Jy      K
```

```
        object          float64 float64
----------------------- ------- -------
2016-03-22T12:30:31.000     1.0    40.0
2016-03-22T12:30:34.000     4.0    41.0
```

Converting between Different Time Representations

In Accessing Times, we saw how to access the time columns/attributes of the **`TimeSeries`** and **`BinnedTimeSeries`** classes. Here we look in more detail at how to manipulate the resulting times.

Converting Times

Since the time column in time series is always a **`Time`** object, it is possible to use the usual attributes on **`Time`** to convert the time to different formats or scales.

Example

To get the times as modified Julian Dates from a minimal time series:

```
>>> from astropy import units as u
>>> from astropy.timeseries import TimeSeries
>>> ts = TimeSeries(time_start='2016-03-22T12:30:31', time_delta=3 *
u.s,
...                 data={'flux': [1., 3., 4., 2., 4.]})
>>> ts.time.mjd
array([57469.52119213, 57469.52122685, 57469.52126157, 57469.5212963
,
       57469.52133102])
```

Or to convert the times to the Temps Atomique International (TAI) scale:

```
>>> ts.time.tai
<Time object: scale='tai' format='isot' value=
['2016-03-22T12:31:07.000' '2016-03-22T12:31:10.000'
 '2016-03-22T12:31:13.000' '2016-03-22T12:31:16.000'
 '2016-03-22T12:31:19.000']>
```

To find the current time scale of the data, you can do:

```
>>> ts.time.scale
'utc'
```

See Time and Dates (astropy.time) for more documentation on how to access and convert times.

Formatting Times

Since the various time columns are **Time** objects, the default format and scale to use for the display of the time series can be changed using the `format` and `scale` attributes.

Example

To change the display of the time series:

```
>>> ts.time.format = 'isot'
>>> ts
<TimeSeries length=5>
          time             flux
         object          float64
----------------------- -------
2016-03-22T12:30:31.000     1.0
2016-03-22T12:30:34.000     3.0
2016-03-22T12:30:37.000     4.0
2016-03-22T12:30:40.000     2.0
2016-03-22T12:30:43.000     4.0
>>> ts.time.format = 'unix'
>>> ts
<TimeSeries length=5>
    time       flux
   object     float64
------------ -------
1458649831.0     1.0
1458649834.0     3.0
1458649837.0     4.0
1458649840.0     2.0
1458649843.0     4.0
```

Times Relative to Other Times

In some cases, it can be useful to use relative rather than absolute times. This can be done by using the **TimeDelta** class instead of the **Time** class, for example, by subtracting a reference time from an existing **Time** object.

Example

To use a relative rather than an absolute time:

```
>>> ts_rel = TimeSeries(time=ts.time - ts.time[0])
>>> ts_rel
<TimeSeries length=5>
         time
        object
----------------------
```

```
                    0.0
 3.4722222222222765e-05
  6.944444444444553e-05
0.00010416666666657193
0.00013888888888879958
```

The `TimeDelta` values can be converted to a different time unit (e.g., second) using:

```
>>> ts_rel.time.to('second')
<Quantity [ 0.,  3.,  6.,  9., 12.] s>
```

Manipulation and Analysis of Time Series

Combining Time Series

The `vstack()` and `hstack()` functions from the `astropy.table` module can be used to stack time series in different ways.

Examples

Time series can be stacked “vertically” or row-wise using the `vstack()` function (although note that sampled time series cannot be combined with binned time series and vice versa):

```
>>> from astropy.table import vstack
>>> from astropy import units as u
>>> from astropy.timeseries import TimeSeries
>>> ts_a = TimeSeries(time_start='2016-03-22T12:30:31',
...                   time_delta=3 * u.s,
...                   data={'flux': [1, 4, 5, 3, 2] * u.mJy})
>>> ts_b = TimeSeries(time_start='2016-03-22T12:50:31',
...                   time_delta=3 * u.s,
...                   data={'flux': [4, 3, 1, 2, 3] * u.mJy})
>>> ts_ab = vstack([ts_a, ts_b])
>>> ts_ab
<TimeSeries length=10>
          time             flux
                           mJy
         object          float64
----------------------- -------
2016-03-22T12:30:31.000     1.0
2016-03-22T12:30:34.000     4.0
2016-03-22T12:30:37.000     5.0
2016-03-22T12:30:40.000     3.0
2016-03-22T12:30:43.000     2.0
```

```
2016-03-22T12:50:31.000     4.0
2016-03-22T12:50:34.000     3.0
2016-03-22T12:50:37.000     1.0
2016-03-22T12:50:40.000     2.0
2016-03-22T12:50:43.000     3.0
```

Note that **vstack()** does not automatically sort, nor get rid of duplicates — this is something you would need to do explicitly afterwards.

Time series can also be combined “horizontally” or column-wise with other tables using the **hstack()** function, though these should not be time series (as having multiple time columns would be confusing):

```
>>> from astropy.table import Table, hstack
>>> data = Table(data={'temperature': [40., 41., 40., 39., 30.] *
u.K})
>>> ts_a_data = hstack([ts_a, data])
>>> ts_a_data
<TimeSeries length=5>
          time             flux  temperature
                           mJy        K
         object          float64    float64
----------------------- ------- -----------
2016-03-22T12:30:31.000     1.0        40.0
2016-03-22T12:30:34.000     4.0        41.0
2016-03-22T12:30:37.000     5.0        40.0
2016-03-22T12:30:40.000     3.0        39.0
2016-03-22T12:30:43.000     2.0        30.0
```

Sorting Time Series

Sorting time series in place can be done using the **sort()** method, as for **Table**:

```
>>> ts = TimeSeries(time_start='2016-03-22T12:30:31',
...                 time_delta=3 * u.s,
...                 data={'flux': [1., 4., 5., 3., 2.]})
>>> ts
<TimeSeries length=5>
          time             flux
         object          float64
----------------------- -------
2016-03-22T12:30:31.000     1.0
2016-03-22T12:30:34.000     4.0
2016-03-22T12:30:37.000     5.0
2016-03-22T12:30:40.000     3.0
2016-03-22T12:30:43.000     2.0
>>> ts.sort('flux')
>>> ts
<TimeSeries length=5>
          time             flux
         object          float64
----------------------- -------
2016-03-22T12:30:31.000     1.0
2016-03-22T12:30:43.000     2.0
2016-03-22T12:30:40.000     3.0
2016-03-22T12:30:34.000     4.0
2016-03-22T12:30:37.000     5.0
```

Resampling

We provide a **aggregate_downsample()** function that can be used to bin values from a time series into bins of equal time, using a custom function (mean, median, etc.). This operation returns a **BinnedTimeSeries**. Note that this is a basic function in the sense that it does not, for example, know how to treat columns with uncertainties differently from other values, and it will blindly apply the custom function specified to all columns.

Example

The following example shows how to use **aggregate_downsample()** to bin a light curve from the Kepler mission into 20 minute bins using a median function. First, we read in the data using:

```
from astropy.timeseries import TimeSeries
from astropy.utils.data import get_pkg_data_filename
example_data =
get_pkg_data_filename('timeseries/kplr010666592-2009131110544_slc.fit
s')
kepler = TimeSeries.read(example_data, format='kepler.fits')
```

(See Reading and Writing Time Series for more details about reading in data). We can then downsample using:

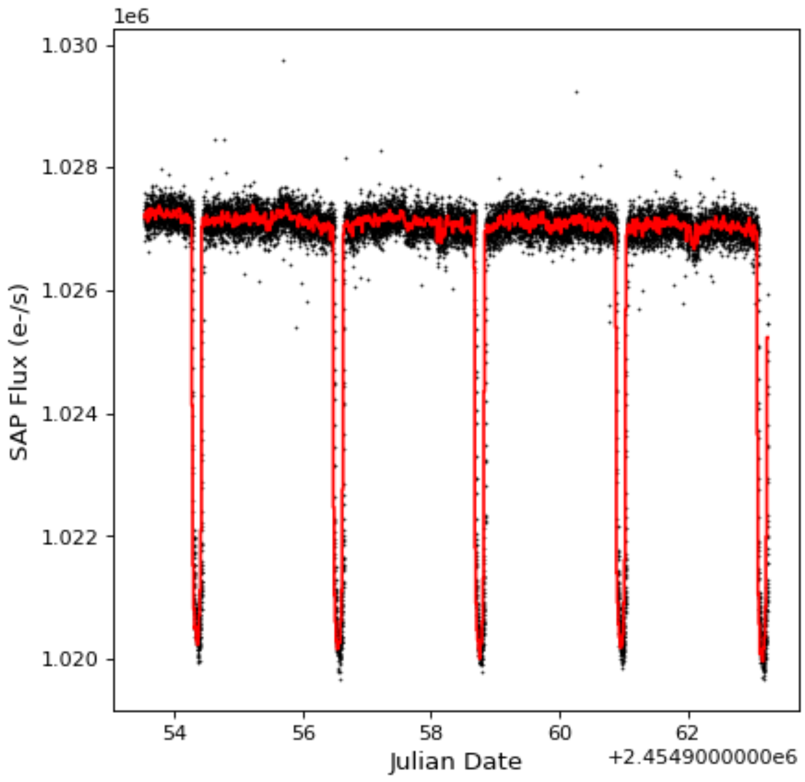
```
import numpy as np
from astropy import units as u
from astropy.timeseries import aggregate_downsample
kepler_binned = aggregate_downsample(kepler, time_bin_size=20 *
u.min, aggregate_func=np.nanmedian)
```

We can take a look at the results:

```
import matplotlib.pyplot as plt
plt.plot(kepler.time.jd, kepler['sap_flux'], 'k.', markersize=1)
plt.plot(kepler_binned.time_bin_start.jd, kepler_binned['sap_flux'],
'r-', drawstyle='steps-pre')
```

```
plt.xlabel('Julian Date')
plt.ylabel('SAP Flux (e-/s)')
```

(png, svg, pdf)

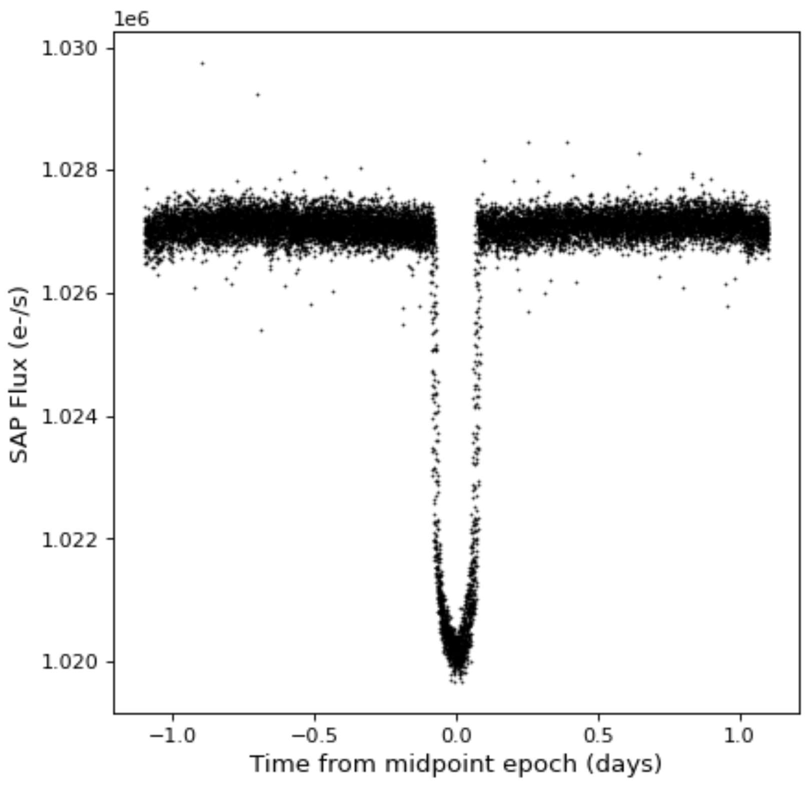


Folding

The **TimeSeries** class has a **fold()** method that can be used to return a new time series with a relative and folded time axis. This method takes the period as a **Quantity**, and optionally takes an epoch as a **Time**, which defines a zero time offset:

```
kepler_folded = kepler.fold(period=2.2 * u.day,
epoch_time='2009-05-02T20:53:40')

plt.plot(kepler_folded.time.jd, kepler_folded['sap_flux'], 'k.',
markersize=1)
plt.xlabel('Time from midpoint epoch (days)')
plt.ylabel('SAP Flux (e-/s)')
```

(png, svg, pdf)



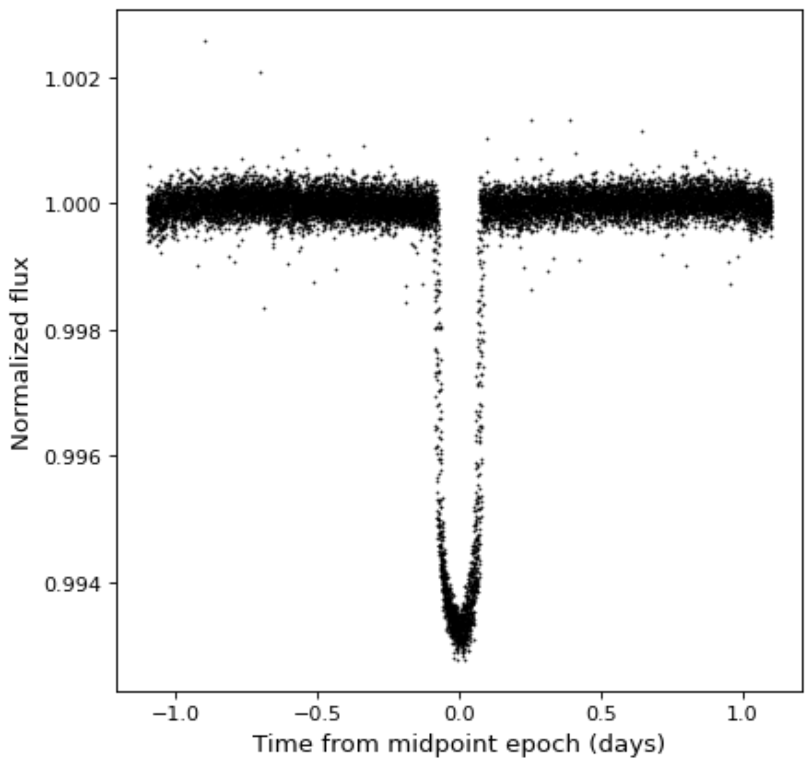
Note that in this example we happened to know the period and midpoint from a previous periodogram analysis. See the example in Time Series (astropy.timeseries) for how you might do this.

Arithmetic

Since `TimeSeries` objects are subclasses of `Table`, they naturally support arithmetic on any of the data columns. As an example, we can take the folded Kepler time series we have seen in previous examples, and normalize it to the sigma-clipped median value.

```
from astropy.stats import sigma_clipped_stats

mean, median, stddev = sigma_clipped_stats(kepler_folded['sap_flux'])

kepler_folded['sap_flux_norm'] = kepler_folded['sap_flux'] / median

plt.plot(kepler_folded.time.jd, kepler_folded['sap_flux_norm'], 'k.',
markersize=1)
plt.xlabel('Time from midpoint epoch (days)')
plt.ylabel('Normalized flux')
```

(png, svg, pdf)



Masking Values in Time Series

Warning

Note that masking does not yet work for columns that have units.

Masking values is done in the same way as for **`Table`** objects (see Masking and Missing Values). The most convenient way to use masking is to initialize a **`TimeSeries`** object using the `masked=True` option.

Example

We start by initializing a **`TimeSeries`** object with `masked=True`:

```
>>> from astropy import units as u
>>> from astropy.timeseries import TimeSeries
>>> ts = TimeSeries(time_start='2016-03-22T12:30:31',
...                 time_delta=3 * u.s,
...                 n_samples=5, masked=True)
```

We can now add some data to our time series:

```
>>> ts['flux'] = [1., -2., 5., -1., 4.]
```

As you can see, some of the values are negative. We can mask these using:

```
>>> ts['flux'].mask = ts['flux'] < 0
>>> ts
<TimeSeries masked=True length=5>
          time             flux
         object          float64
----------------------- -------
2016-03-22T12:30:31.000     1.0
2016-03-22T12:30:34.000      --
2016-03-22T12:30:37.000     5.0
2016-03-22T12:30:40.000      --
2016-03-22T12:30:43.000     4.0
```

We can also access the mask values:

```
>>> ts['flux'].mask
array([False,  True, False,  True, False]...)
```

Masks are column-based, so masking a single cell does not mask the whole row. Having masked cells then allows functions that normally understand masked values and operate on columns to ignore the masked entries:

```
>>> import numpy as np
>>> np.min(ts['flux'])
1.0
>>> np.ma.median(ts['flux'])
4.0
```

Interfacing with the Pandas Package

The **`astropy.timeseries`** package is not the only package to provide functionality related to time series. Another notable package is pandas, which provides a **`pandas.DataFrame`** class. The main benefits of **`astropy.timeseries`** in the context of astronomical research are the following:

- The time column is a **`Time`** object that supports very high precision representation of times, and makes it easy to convert between different time scales and formats (e.g., ISO 8601 timestamps, Julian Dates, and so on).
- The data columns can include **`Quantity`** objects with units.
- The **`BinnedTimeSeries`** class includes variable-width time bins.
- There are built-in readers for common time series file formats, as well as the

ability to define custom readers/writers.

Nevertheless, there are cases where using pandas **DataFrame** objects might make sense, so we provide methods to convert to/from **DataFrame** objects.

Example

Consider a concise example starting from a **DataFrame**:

```
>>> import pandas
>>> import numpy as np
>>> df = pandas.DataFrame()
>>> df['a'] = [1, 2, 3]
>>> times = np.array(['2015-07-04', '2015-07-05', '2015-07-06'],
dtype=np.datetime64)
>>> df.set_index(pandas.DatetimeIndex(times), inplace=True)
>>> df
    a
2015-07-04  1
2015-07-05  2
2015-07-06  3
```

We can convert this to an `astropy` **TimeSeries** using **from_pandas()**:

```
>>> from astropy.timeseries import TimeSeries
>>> ts = TimeSeries.from_pandas(df)
>>> ts
<TimeSeries length=3>
             time                a
            object             int64
----------------------------- -----
2015-07-04T00:00:00.000000000     1
2015-07-05T00:00:00.000000000     2
2015-07-06T00:00:00.000000000     3
```

Converting to **DataFrame** can also be done with **to_pandas()**:

```
>>> ts['b'] = [1.2, 3.4, 5.4]
>>> df_new = ts.to_pandas()
>>> df_new
            a    b
time
2015-07-04  1  1.2
2015-07-05  2  3.4
2015-07-06  3  5.4
```

Missing values in the time column are supported and correctly converted to a pandas' NaT object:

```
>>> ts.time[2] = np.nan
>>> ts
<TimeSeries length=3>
             time                a       b
            object             int64 float64
----------------------------- ----- -------
2015-07-04T00:00:00.000000000     1     1.2
2015-07-05T00:00:00.000000000     2     3.4
                           --     3     5.4
>>> df_missing = ts.to_pandas()
>>> df_missing
           a    b
time
2015-07-04  1  1.2
2015-07-05  2  3.4
NaT         3  5.4
```

Periodogram Algorithms

Lomb-Scargle Periodograms

The Lomb-Scargle periodogram (after Lomb [1], and Scargle [2]) is a commonly used statistical tool designed to detect periodic signals in unevenly spaced observations. The `LombScargle` class is a unified interface to several implementations of the Lomb-Scargle periodogram, including a fast *O[NlogN]* implementation following the algorithm presented by Press & Rybicki [3].

The code here is adapted from the astroml package ([4], [5]) and the gatspy package ([6], [7]). For a detailed practical discussion of the Lomb-Scargle periodogram, with code examples based on `astropy`, see *Understanding the Lomb-Scargle Periodogram* [11], with associated code at https://github.com/jakevdp/PracticalLombScargle/.

Basic Usage

Note

All frequencies in `LombScargle` are **not** angular frequencies, but rather frequencies of oscillation (i.e., number of cycles per unit time).

The Lomb-Scargle periodogram is designed to detect periodic signals in unevenly spaced observations.

Example

To detect periodic signals in unevenly spaced observations, consider the following data:

```
>>> import numpy as np
>>> rand = np.random.RandomState(42)
>>> t = 100 * rand.rand(100)
>>> y = np.sin(2 * np.pi * t) + 0.1 * rand.randn(100)
```

These are 100 noisy measurements taken at irregular times, with a frequency of 1 cycle per unit time.

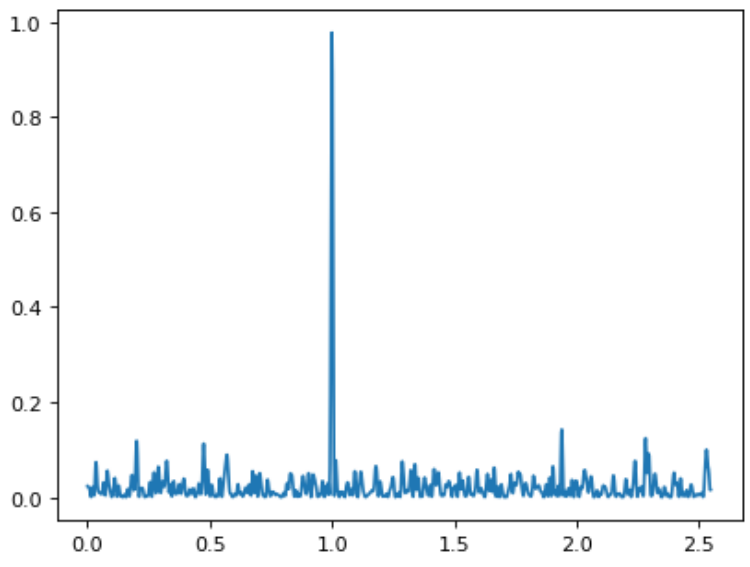
The Lomb-Scargle periodogram, evaluated at frequencies chosen automatically based on the input data, can be computed as follows using the `LombScargle` class:

```
>>> from astropy.timeseries import LombScargle
>>> frequency, power = LombScargle(t, y).autopower()
```

Plotting the result with Matplotlib gives:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(frequency, power)
```

(png, svg, pdf)



The periodogram shows a clear spike at a frequency of 1 cycle per unit time, as we would expect from the data we constructed.

Measurement Uncertainties

The `LombScargle` interface can also handle data with measurement uncertainties.

Example

If all uncertainties are the same, you can pass a scalar:

```
>>> dy = 0.1
>>> frequency, power = LombScargle(t, y, dy).autopower()
```

If uncertainties vary from observation to observation, you can pass them as an array:

```
>>> dy = 0.1 * (1 + rand.rand(100))
>>> y = np.sin(2 * np.pi * t) + dy * rand.randn(100)
>>> frequency, power = LombScargle(t, y, dy).autopower()
```

Gaussian uncertainties are assumed, and `dy` here specifies the standard deviation (not the variance).

Periodograms and Units

The **`LombScargle`** interface properly handles **`Quantity`** objects with units attached, and will validate the inputs to make sure units are appropriate.

Example

To use the **`LombScargle`** for **`Quantity`** objects with units attached:

```
>>> import astropy.units as u
>>> t_days = t * u.day
>>> y_mags = y * u.mag
>>> dy_mags = y * u.mag
>>> frequency, power = LombScargle(t_days, y_mags,
dy_mags).autopower()
>>> frequency.unit
Unit("1 / d")
>>> power.unit
Unit(dimensionless)
```

We see that the output is dimensionless, which is always the case for the standard normalized periodogram (for more on normalizations, see Periodogram Normalizations below).

Specifying the Frequency

With the **`autopower()`** method used above, a heuristic is applied to select a suitable frequency grid. By default, the heuristic assumes that the width of peaks is inversely proportional to the observation baseline, and that the maximum frequency is a factor of five larger than the so-called “average Nyquist frequency,” with computation based on the average observation spacing.

This heuristic is not universally useful, as the frequencies probed by irregularly sampled data can be much higher than the average Nyquist frequency. For this reason, the heuristic can be tuned through keywords passed to the **autopower()** method.

Example

To tune the heuristic using keywords passed to the **autopower()** method:

```
>>> frequency, power = LombScargle(t, y,
dy).autopower(nyquist_factor=2)
>>> len(frequency), frequency.min(), frequency.max()
(500, 0.0010189890448009111, 1.0179700557561102)
```

Here the highest frequency is two times the average Nyquist frequency. If we increase the `nyquist_factor`, we can probe higher frequencies:

```
>>> frequency, power = LombScargle(t, y,
dy).autopower(nyquist_factor=10)
>>> len(frequency), frequency.min(), frequency.max()
(2500, 0.0010189890448009111, 5.0939262349597545)
```

Alternatively, we can use the **power()** method to evaluate the periodogram at a user-specified set of frequencies:

```
>>> frequency = np.linspace(0.5, 1.5, 1000)
>>> power = LombScargle(t, y, dy).power(frequency)
```

Note that the fastest Lomb-Scargle implementation requires regularly spaced frequencies; if frequencies are irregularly spaced, a slower method will be used instead.

Frequency Grid Spacing

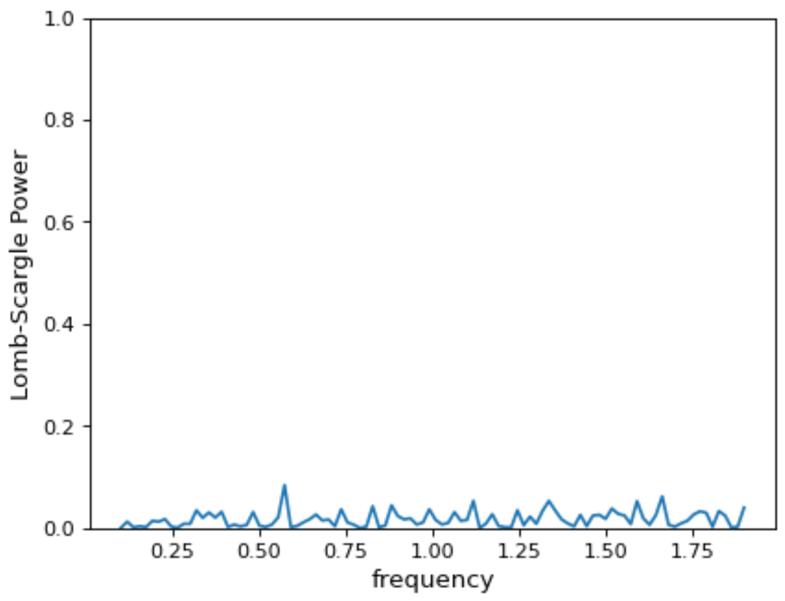
One common issue with user-specified frequencies is inadvertently choosing too coarse a grid, such that significant peaks lie between grid points and are missed entirely.

Example

Imagine you chose to evaluate your periodogram at 100 points:

```
>>> frequency = np.linspace(0.1, 1.9, 100)
>>> power = LombScargle(t, y, dy).power(frequency)
>>> plt.plot(frequency, power)
```

(png, svg, pdf)

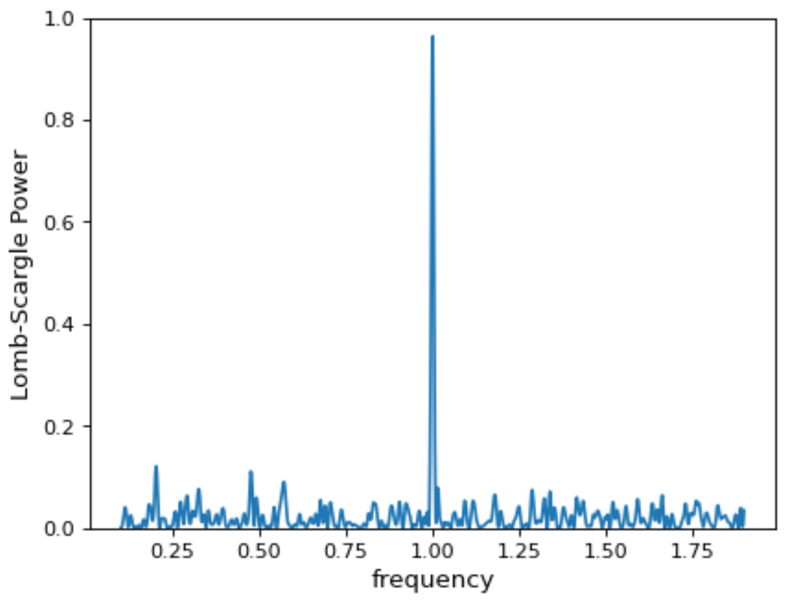


From this plot alone, you might conclude that no clear periodic signal exists in the data. But this conclusion is in error: there is in fact a strong periodic signal, but the periodogram peak falls in the gap between the chosen grid points!

A more reliable approach is to use the frequency heuristic to decide on the appropriate grid spacing, optionally passing a minimum and maximum frequency to the **`autopower()`** method:

```
>>> frequency, power = LombScargle(t, y,
dy).autopower(minimum_frequency=0.1,
...
maximum_frequency=1.9)
>>> len(frequency)
884
>>> plt.plot(frequency, power)
```

(png, svg, pdf)



With a finer grid (here 884 points between 0.1 and 1.9), it is clear that there is a very strong periodic signal in the data.

By default, the heuristic aims to have roughly five grid points across each significant periodogram peak; this can be increased by changing the `samples_per_peak` argument:

```
>>> frequency, power = LombScargle(t, y,
dy).autopower(minimum_frequency=0.1,
...
maximum_frequency=1.9,
...
samples_per_peak=10)
>>> len(frequency)
1767
```

Keep in mind that the width of the peak scales inversely with the baseline of the observations (i.e., the difference between the maximum and minimum time), and the required number of grid points will scale linearly with the size of the baseline.

The Lomb-Scargle Model

The Lomb-Scargle periodogram fits a sinusoidal model to the data at each frequency, with a larger power reflecting a better fit. With this in mind, it is often helpful to plot the best-fit sinusoid over the phased data.

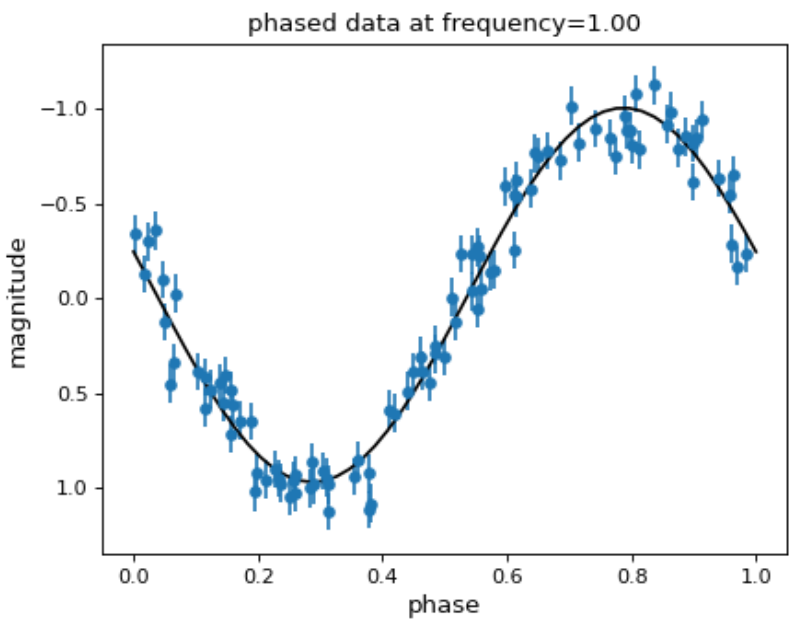
Example

This best-fit sinusoid can be computed using the `model()` method of the `LombScargle` object:

```
>>> best_frequency = frequency[np.argmax(power)]
>>> t_fit = np.linspace(0, 1)
>>> ls = LombScargle(t, y, dy)
>>> y_fit = ls.model(t_fit, best_frequency)
```

We can then phase the data and plot the Lomb-Scargle model fit:

(png, svg, pdf)



The best-fit model parameters can be computed with the **model_parameters()** method of the **LombScargle** object at a given frequency:

```
>>> theta = ls.model_parameters(best_frequency)
>>> theta.round(2)
array([-0.02,  1.05,  0.07])
```

These parameters \(\vec{\theta}\) are fit using the following model:

\[y(t; f, \vec{\theta}) = \theta_0 + \sum_{n=1}^{\tt nterms} [\theta_{2n-1}\sin(2\pi n f t) + \theta_{2n}\cos(2\pi n f t)]\]

The model can be constructed from these parameters by computing the associated **offset()**, which accounts for the pre-centering of data (i.e., the `center_data` argument), and **design_matrix()**, which computes the sine and cosine terms for you:

```
>>> offset = ls.offset()
>>> design_matrix = ls.design_matrix(best_frequency, t_fit)
>>> np.allclose(y_fit, offset + design_matrix.dot(theta))
True
```

Additional Arguments

On initialization, `LombScargle` takes a few additional arguments which control the model for the data:

- `center_data` (`True` by default) controls whether the `y` values are pre-centered before the algorithm fits the data. The only time it is really warranted to change the default is if you are computing the periodogram of a sequence of constant values to, for example, estimate the window power spectrum for a series of observations.
- `fit_mean` (`True` by default) controls whether the model fits for the mean of the data, rather than assuming the mean is zero. When `fit_mean=True` , the periodogram is more robust than the original Lomb-Scargle formalism, particularly in the case of smaller sample sizes and/or data with nontrivial selection bias. In the literature, this model has variously been called the *date-compensated discrete Fourier transform*, the *floating-mean periodogram*, the *generalized Lomb-Scargle method*, and likely other names as well.
- `nterms` (`1` by default) controls how many Fourier terms are used in the model. As seen above, the standard Lomb-Scargle periodogram is equivalent to a single-term sinusoidal fit to the data at each frequency; the generalization is to expand this to a truncated Fourier series with multiple frequencies. While this can be very useful in some cases, in others the additional model complexity can lead to spurious periodogram peaks that outweigh the benefit of the more flexible model.

Periodogram Normalizations

There are several normalizations of the Lomb-Scargle periodogram found in the literature. `LombScargle` makes four options available via the `normalization` argument: `normalization='standard'` (the default), `normalization='model'` , `normalization='log'` , and `normalization='psd'` . These normalizations can be thought of in terms of least-squares fits around a constant reference model M_{ref} and a periodic model $M(f)$ at each frequency, with best-fit sum of residuals that we will denote by χ^2_{ref} and $\chi^2(f)$ respectively.

Standard Normalization

The default, the standard normalized periodogram is normalized by the residuals of the data around the constant reference model:

$$P_{standard}(f) = \frac{\chi^2_{ref} - \chi^2(f)}{\chi^2_{ref}}$$

This form of the normalization (`normalization='standard'`) is the default choice used in `LombScargle`. The resulting power P is a dimensionless quantity that lies in the range $0 \le P \le 1$.

Model Normalization

Alternatively, the periodogram is sometimes normalized instead by the residuals around the periodic model:

$$P_{model}(f) = \frac{\chi^2_{ref} - \chi^2(f)}{\chi^2(f)}$$

This form of the normalization can be specified with `normalization='model'`. As above, the resulting power is a dimensionless quantity that lies in the range $0 \le P \le \infty$.

Logarithmic Normalization

Another form of normalization is to scale the periodogram logarithmically:

$$P_{log}(f) = \log \frac{\chi^2_{ref}}{\chi^2(f)}$$

This normalization can be specified with `normalization='log'`, and the resulting power is a dimensionless quantity in the range $0 \le P \le \infty$.

PSD Normalization (Unnormalized)

Finally, it is sometimes useful to compute an unnormalized periodogram (`normalization='psd'`):

$$P_{psd}(f) = \frac{1}{2}\left(\chi^2_{ref} - \chi^2(f)\right)$$

Which, in the case of no-uncertainty, will have units `y.unit ** 2`. This normalization is constructed to be comparable to the standard Fourier power spectral density (PSD):

```
>>> ls = LombScargle(t_days, y_mags, normalization='psd')
>>> frequency, power = ls.autopower()
>>> power.unit
Unit("mag2")
```

Note, however, that the `normalization='psd'` result only has these units *if uncertainties are not specified*. In the presence of uncertainties, even the unnormalized PSD periodogram will be dimensionless; this is due to the scaling of data by uncertainty within the Lomb-Scargle computation:

```
>>> # with uncertainties, PSD power is unitless
>>> ls = LombScargle(t_days, y_mags, dy_mags, normalization='psd')
>>> frequency, power = ls.autopower()
>>> power.unit
Unit(dimensionless)
```

The equivalence of the PSD-normalized periodogram and the Fourier PSD in the unnormalized, no-uncertainty case can be confirmed by comparing results directly for uniformly sampled inputs.

We will first define a convenience function to compute the basic Fourier periodogram for uniformly sampled quantities:

```
>>> def fourier_periodogram(t, y):
...     N = len(t)
...     frequency = np.fft.fftfreq(N, t[1] - t[0])
...     y_fft = np.fft.fft(y.value) * y.unit
...     positive = (frequency > 0)
...     return frequency[positive], (1. / N) * abs(y_fft[positive])
** 2
```

Next we compute the two versions of the PSD from uniformly sampled data:

```
>>> t_days = np.arange(100) * u.day
>>> y_mags = rand.randn(100) * u.mag
>>> frequency, PSD_fourier = fourier_periodogram(t_days, y_mags)
>>> ls = LombScargle(t_days, y_mags, normalization='psd')
>>> PSD_LS = ls.power(frequency)
```

Examining the results, we see that the two outputs match:

```
>>> u.allclose(PSD_fourier, PSD_LS)
True
```

This equivalence is one reason that the Lomb-Scargle periodogram is considered to be an extension of the Fourier PSD.

For more information on the statistical properties of these normalizations, see, for example, Baluev 2008 [8].

Peak Significance and False Alarm Probabilities

Note

Interpretation of Lomb-Scargle peak significance via false alarm probabilities is a subtle subject, and the quantities computed below are commonly misinterpreted or misused. For a detailed discussion of periodogram peak significance, see [11].

When using the Lomb-Scargle periodogram to decide whether a signal contains a periodic component, an important consideration is the significance of the periodogram peak. This significance is usually expressed in terms of a false alarm probability, which encodes the probability of measuring a peak of a given height (or higher) conditioned on the assumption that the data consists of Gaussian noise with no periodic component.

Example

To use the Lomb-Scargle periodogram to decide if our signal contains a periodic component, we can start by simulating 60 observations of a sine wave with noise:

```
>>> t = 100 * rand.rand(60)
>>> dy = 1.0
>>> y = np.sin(2 * np.pi * t) + dy * rand.randn(60)
>>> ls = LombScargle(t, y, dy)
>>> freq, power = ls.autopower()
>>> print(power.max())
0.33814001958188855
```

The peak of the periodogram has a value of 0.33, but how significant is this peak? We can address this question using the `false_alarm_probability()` method:

```
>>> ls.false_alarm_probability(power.max())
0.0043217866919174324
```

What this tells us is that under the assumption that there is no periodic signal in the data, we will observe a peak this high or higher approximately 0.4% of the time, which gives a strong indication that a periodic signal is present in the data.

Note

Users must interpret this probability carefully: it is a measurement conditioned on the assumption of the null hypothesis of no signal; in symbols, you might write \(P({\rm data} \mid {\rm noise-only})\).

Although it may seem like this quantity could be interpreted with a statement such as “there is an 0.4% chance that this data is noise only,” this is *not* a correct statement; in symbols, this statement describes the quantity \(P({\rm noise-only} \mid {\rm data})\), and in general \(P(A\mid B) \ne P(B\mid A)\).

See [11] for a more detailed discussion of such caveats.

We might also wish to compute the required peak height to attain any given false alarm probability, which can be done with the `false_alarm_level()` method:

```
>>> probabilities = [0.1, 0.05, 0.01]
>>> ls.false_alarm_level(probabilities)
array([0.25446627, 0.27436154, 0.31716182])
```

This tells us that to attain a 10% false alarm probability requires the highest periodogram peak to be approximately 0.25; 5% requires 0.27, and 1% requires 0.32.

False Alarm Approximations

Although the false alarm probability at any particular frequency is analytically computable, there is no closed-form analytic expression for the more relevant quantity of the false alarm level of the *highest* peak in a particular periodogram. This must be either determined through bootstrap simulations, or approximated by various means.

`astropy` provides four options for approximating the false alarm probability, which can be chosen using the `method` keyword:

- `method="baluev"` (the default) implements the approximation proposed by Baluev 2008 [8], which employs extreme value statistics to compute an upper bound of the false alarm probability for the alias-free case. Experiments show that the bound is also useful even for highly aliased observing patterns.

```
>>> ls.false_alarm_probability(power.max(), method='baluev')
0.0043217866919174324
```

- `method="bootstrap"` implements a bootstrap simulation: effectively it computes many Lomb-Scargle periodograms on simulated data at the same observation times. The bootstrap approach can very accurately determine the false alarm probability, but is very computationally expensive. To estimate the level corresponding to a false alarm probability \(P_{false}\), it requires on order \(n_{boot} \approx 10/P_{false}\) individual periodograms to be computed for the dataset.

```
>>> ls.false_alarm_probability(power.max(), method='bootstrap')
0.0030000000000000027
```

- `method="davies"` is related to the Baluev method, but loses accuracy at large false alarm probabilities.

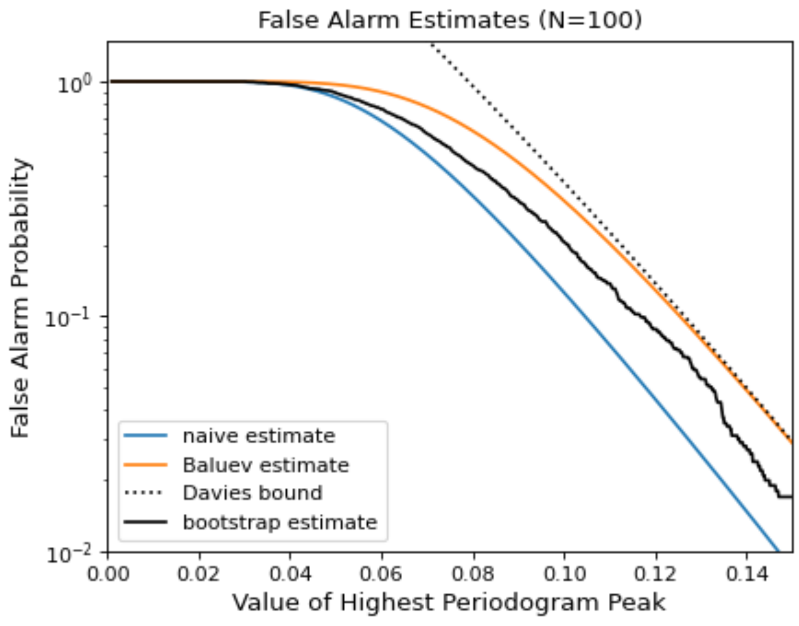
```
>>> ls.false_alarm_probability(power.max(), method='davies')
0.0043311525763707216
```

- `method="naive"` is a basic method based on the assumption that well-separated areas in the periodogram are independent. In general, it provides a very poor estimate of the false alarm probability and should not be used in practice, but is included for completeness.

```
>>> ls.false_alarm_probability(power.max(), method='naive')
0.0011693992470136049
```

The following figure compares these false alarm estimates at a range of peak heights for 100 observations with a heavily aliased observing pattern:

(png, svg, pdf)



In general, users should use the bootstrap approach when computationally feasible, and the Baluev approach otherwise.

In all of this, it is important to keep in mind a few caveats:

- False alarm probabilities are computed relative to a particular set of observing times, and a particular choice of frequency grid.
- False alarm probabilities are conditioned upon the null hypothesis of data with no periodic component, and in particular say nothing quantitative about whether the data are actually consistent with a periodic model.
- False alarm probabilities are not related to the question of whether the highest peak in a periodogram is the *correct* peak, and in particular are not especially useful in the case of observations with a strong aliasing pattern.

For a detailed discussion of these caveats and others when computing and interpreting false alarm probabilities, please refer to [11].

Periodogram Algorithms

The **`LombScargle`** class makes available several complementary implementations of the Lomb-Scargle periodogram, which can be selected using the `method` keyword of the Lomb-Scargle power. By design all methods will return the same results (some approximate), and each has its advantages and disadvantages.

For example, to compute a periodogram using the Fast Chi-squared method of Palmer (2009) [9], you can specify `method='fastchi2'`:

```
>>> frequency, power = LombScargle(t, y).autopower(method='fastchi2')
```

There are currently six methods available in the package:

`method='auto'`

The `auto` method is the default, and will attempt to select the best option from the following methods using heuristics driven by the input data.

`method='slow'`

The `slow` method is a pure Python implementation of the original Lomb-Scargle periodogram ([1], [2]), enhanced to account for observational noise, and to allow a floating mean (sometimes called the *generalized periodogram*; see [10]). The method is not particularly fast, scaling approximately as \(O[NM]\) for \(N\) data points and \(M\) frequencies.

`method='cython'`

The `cython` method is a Cython implementation of the same algorithm used for `method='slow'`. It is slightly faster than the pure Python implementation, but much more memory-efficient as the size of the inputs grow. The computational scaling is approximately \(O[NM]\) for \(N\) data points and \(M\) frequencies.

`method='scipy'`

The `scipy` method wraps the C implementation of the original Lomb-Scargle periodogram which is available in **`scipy.signal.lombscargle()`**. This is slightly faster than the `slow` method, but does not allow for errors in data or extensions such as the floating mean. The scaling is approximately \(O[NM]\) for \(N\) data points and \(M\) frequencies.

`method='fast'`

The `fast` method is a pure Python implementation of the fast periodogram of Press & Rybicki [3]. It uses an *extrapolation* approach to approximate the periodogram frequencies using a fast Fourier transform. As with the `slow` method, it can handle data errors and floating mean. The scaling is approximately \(O[N\log M]\) for \(N\) data points and \(M\) frequencies. The fast algorithm trades accuracy for speed, and produces a close approximation to the true periodogram. In particular, you may observe powers less than zero in some cases.

`method='chi2'`

The `chi2` method is a pure Python implementation based on matrix algebra (see [7]). It utilizes the fact that the Lomb-Scargle periodogram at each frequency is equivalent to the least-squares fit of a sinusoid to the data. The advantage of the `chi2` method is that it allows extensions of the periodogram to multiple Fourier terms, specified by the `nterms` parameter. For the standard problem, it is slightly slower than `method='slow'` and scales as $O[n_fNM]$ for N data points, M frequencies, and n_f Fourier terms.

`method='fastchi2'`

The Fast Chi-squared method of Palmer (2009) [9] is equivalent to the `chi2` method, but the matrices are constructed using an FFT-based approach similar to that of the `fast` method. The result is a relatively efficient periodogram (though not nearly as efficient as the `fast` method) which can be extended to multiple terms. The scaling is approximately $O[n_f(M + N\log M)]$ for N data points, M frequencies, and n_f Fourier terms.

Summary

The following table summarizes the features of the above algorithms:

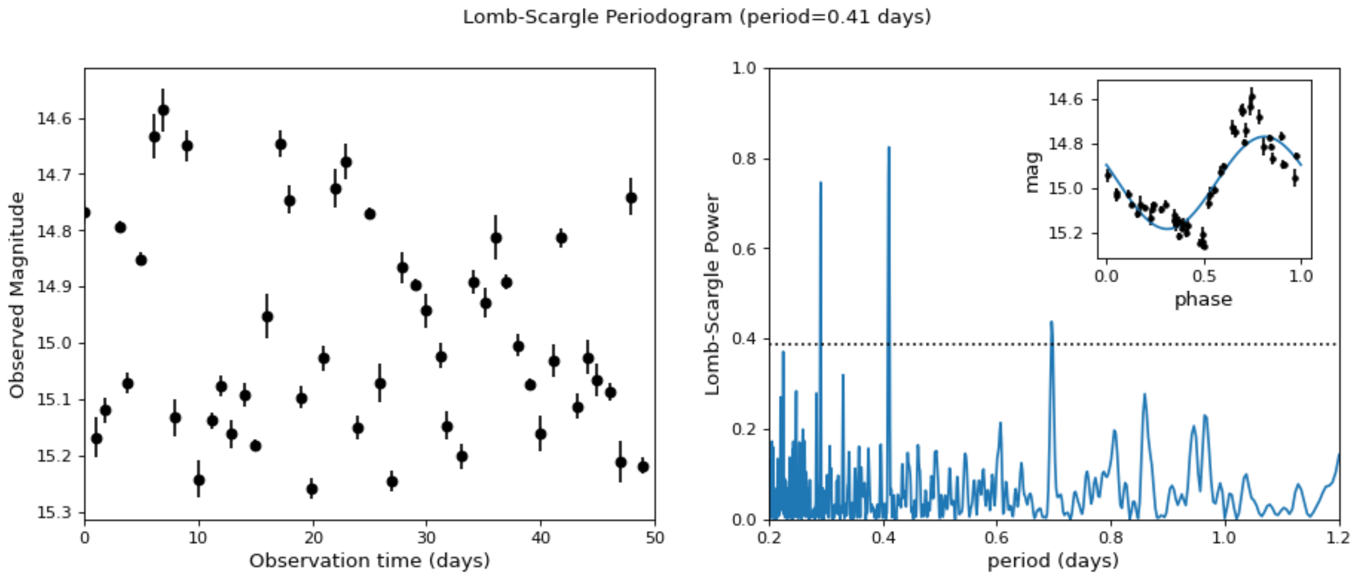
Method	Computational Scaling	Observational Uncertainties	Bias Term (Floating Mean)	Multiple Terms
`"slow"`	$O[NM]$	Yes	Yes	No
`"cython"`	$O[NM]$	Yes	Yes	No
`"scipy"`	$O[NM]$	No	No	No
`"fast"`	$O[N\log M]$	Yes	Yes	No
`"chi2"`	$O[n_fNM]$	Yes	Yes	Yes
`"fastchi2"`	$O[n_f(M + N\log M)]$	Yes	Yes	Yes

In the Computational Scaling column, N is the number of data points, M is the number of frequencies, and n_f is the number of Fourier terms for a multi-term fit.

RR Lyrae Example

An example of computing the periodogram for a more realistic dataset is shown in the following figure. The data here consists of 50 nightly observations of a simulated RR Lyrae-like variable star, with a lightcurve shape that is more complicated than a simple sine wave:

(png, svg, pdf)



The dotted line shows the periodogram level corresponding to a maximum peak false alarm probability of 1%. This example demonstrates that for irregularly sampled data, the Lomb-Scargle periodogram can be sensitive to frequencies higher than the average Nyquist frequency: the above data are sampled at an average rate of roughly one observation per night, and the periodogram relatively cleanly reveals the true period of 0.41 days.

Still, the periodogram has many spurious peaks, which are due to several factors:

1. Errors in observations lead to leakage of power from the true peaks.
2. The signal is not a perfect sinusoid, so additional peaks can indicate higher frequency components in the signal.
3. The observations take place only at night, meaning that the survey window has non-negligible power at a frequency of 1 cycle per day. Thus we expect aliases to appear at \(f_{\rm alias} = f_{\rm true} + n f_{\rm window}\) for integer values of \(n\). With a true period of 0.41 days and a 1-day signal in the observing window, the \(n=+1\) and \(n=-1\) aliases to lie at periods of 0.29 and 0.69 days, respectively: these aliases are prominent in the above plot.

The interaction of these effects means that in practice there is no absolute guarantee that the highest peak corresponds to the best frequency, and results must be interpreted carefully. For a detailed discussion of these effects, see [11].

Literature References

1(1,2) Lomb, N.R. *Least-squares frequency analysis of unequally spaced data*. Ap&SS 39 pp. 447-462 (1976)

2(1,2) Scargle, J. D. *Studies in astronomical time series analysis. II - Statistical aspects of spectral analysis of unevenly spaced data*. ApJ 1:263 pp. 835-853 (1982)

3(1,2) Press W.H. and Rybicki, G.B, *Fast algorithm for spectral analysis of unevenly sampled data*. ApJ 1:338, p. 277 (1989)

[4] Vanderplas, J., Connolly, A. Ivezic, Z. & Gray, A. *Introduction to astroML: Machine learning for astrophysics*. Proceedings of the Conference on Intelligent Data Understanding (2012)

[5] Vanderplas, J., Connolly, A. Ivezic, Z. & Gray, A. *Statistics, Data Mining and Machine Learning in Astronomy*. Princeton Press (2014)}

[6] VanderPlas, J. *Gatspy: General Tools for Astronomical Time Series in Python* (2015) https://zenodo.org/record/14833

7(1,2) VanderPlas, J. & Ivezic, Z. *Periodograms for Multiband Astronomical Time Series*. ApJ 812.1:18 (2015)

8(1,2) Baluev, R.V. *Assessing Statistical Significance of Periodogram Peaks* MNRAS 385, 1279 (2008)

9(1,2) Palmer, D. *A Fast Chi-squared Technique for Period Search of Irregularly Sampled Data*. ApJ 695.1:496 (2009)

[10] Zechmeister, M. and Kurster, M. *The generalised Lomb-Scargle periodogram. A new formalism for the floating-mean and Keplerian periodograms*, A&A 496, 577-584 (2009)

11(1,2,3,4,5) VanderPlas, J. *Understanding the Lomb-Scargle Periodogram* ApJS 236.1:16 (2018) https://ui.adsabs.harvard.edu/abs/2018ApJS..236…16V

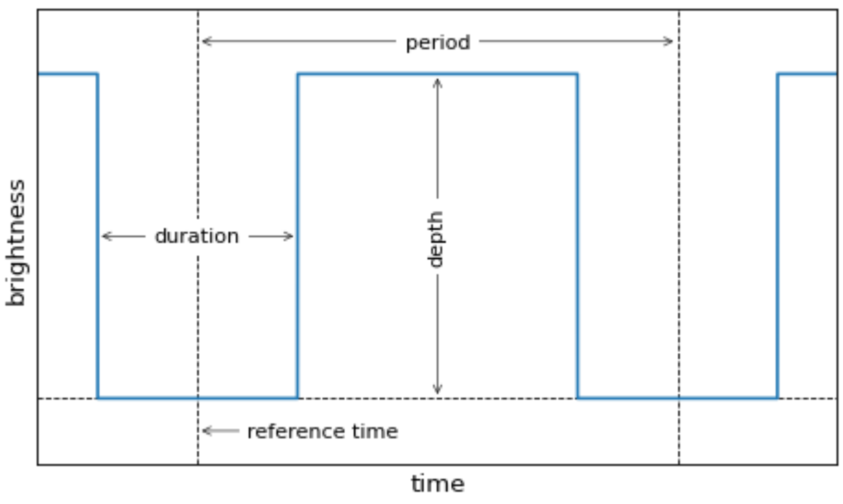
Box Least Squares (BLS) Periodogram

The “box least squares” (BLS) periodogram [1] is a statistical tool used for detecting transiting exoplanets and eclipsing binaries in time series photometric data. The main interface to this implementation is the `BoxLeastSquares` class.

Mathematical Background

The BLS method finds transit candidates by modeling a transit as a periodic upside down top hat with four parameters: period, duration, depth, and a reference time. In this implementation, the reference time is chosen to be the mid-transit time of the first transit in the observational baseline. These parameters are shown in the following sketch:

(png, svg, pdf)



Assuming that the uncertainties on the measured flux are known, independent, and Gaussian, the maximum likelihood in-transit flux can be computed as

$$y_\mathrm{in} = \frac{\sum_\mathrm{in} y_n/{\sigma_n}^2}{\sum_\mathrm{in} 1/{\sigma_n}^2}$$

where y_n are the brightness measurements, σ_n are the associated uncertainties, and both sums are computed over the in-transit data points.

Similarly, the maximum likelihood out-of-transit flux is

$$y_\mathrm{out} = \frac{\sum_\mathrm{out} y_n/{\sigma_n}^2}{\sum_\mathrm{out} 1/{\sigma_n}^2}$$

where these sums are over the out-of-transit observations. Using these results, the log likelihood of a transit model (maximized over depth) at a given period P, duration τ, and reference time t_0 is

$$\log \mathcal{L}(P,\,\tau,\,t_0) = -\frac{1}{2}\,\sum_\mathrm{in}\frac{(y_n-y_\mathrm{in})^2}{{\sigma_n}^2} -\frac{1}{2}\,\sum_\mathrm{out}\frac{(y_n-y_\mathrm{out})^2}{{\sigma_n}^2} + \mathrm{constant}$$

This equation might be familiar because it is proportional to the “chi squared” χ^2 for this model and this is a direct consequence of our assumption of Gaussian uncertainties.

This χ^2 is called the “signal residue” by [1], so maximizing the log likelihood over duration and reference time is equivalent to computing the box least squares spectrum from [1].

In practice, this is achieved by finding the maximum likelihood model over a grid in duration and reference time as specified by the `durations` and `oversample` parameters for the **`power`** method.

Behind the scenes, this implementation minimizes the number of required calculations by pre-binning the observations onto a fine grid following [1] and [2].

Basic Usage

The transit periodogram takes as input time series observations where the timestamps `t` and the observations `y` (usually brightness) are stored as `numpy` arrays or **`Quantity`** objects. If known, error bars `dy` can also optionally be provided.

Example

To evaluate the periodogram for a simulated data set:

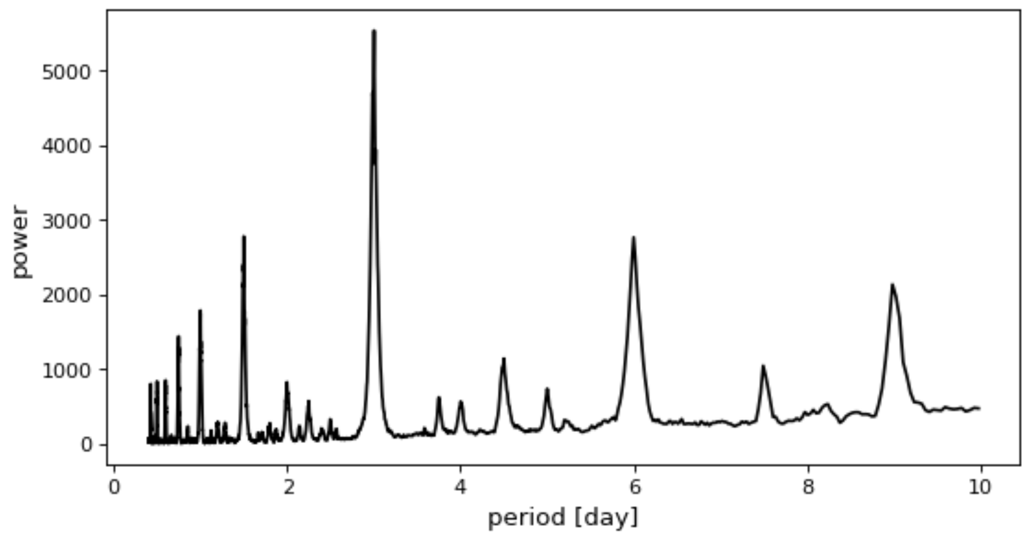
```
>>> import numpy as np
>>> import astropy.units as u
>>> from astropy.timeseries import BoxLeastSquares
>>> np.random.seed(42)
>>> t = np.random.uniform(0, 20, 2000)
>>> y = np.ones_like(t) - 0.1*((t%3)<0.2) +
0.01*np.random.randn(len(t))
>>> model = BoxLeastSquares(t * u.day, y, dy=0.01)
>>> periodogram = model.autopower(0.2)
```

The output of the **`astropy.timeseries.BoxLeastSquares.autopower`** method is a **`BoxLeastSquaresResults`** object with several useful attributes, the most useful of which are generally the `period` and `power` attributes.

This result can be plotted using matplotlib:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(periodogram.period, periodogram.power)
```

(png, svg, pdf)



In this figure, you can see the peak at the correct period of three days.

Objectives

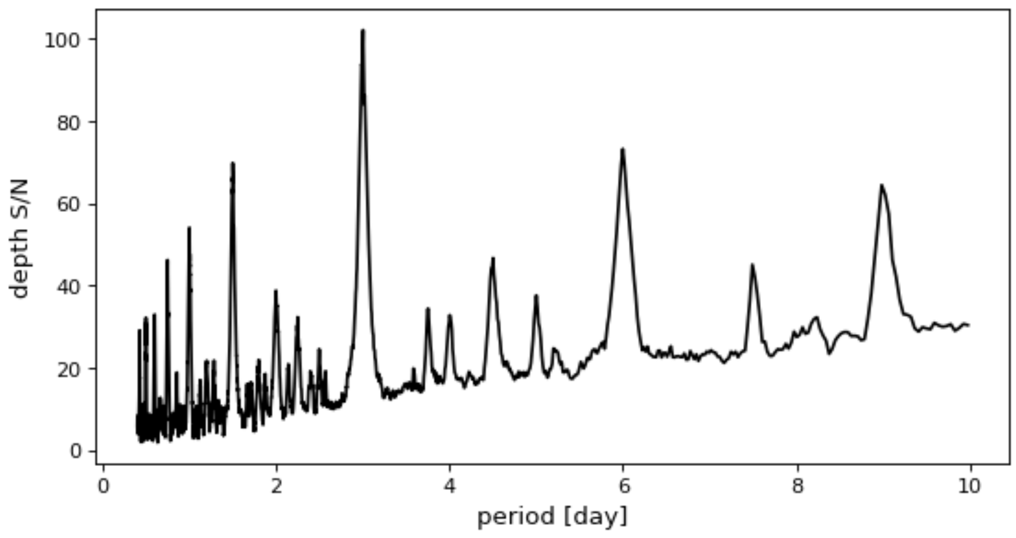
By default, the **power** method computes the log likelihood of the model fit and maximizes over reference time and duration. It is also possible to use the signal-to-noise ratio with which the transit depth is measured as an objective function.

Example

To compute the log likelihood of the model fit, call **power** or **autopower** with `objective='snr'` as follows:

```
>>> model = BoxLeastSquares(t * u.day, y, dy=0.01)
>>> periodogram = model.autopower(0.2, objective="snr")
```

(png, svg, pdf)



This objective will generally produce a periodogram that is qualitatively similar to the log likelihood spectrum, but it has been used to improve the reliability of transit search in the presence of correlated noise.

Period Grid

The transit periodogram is always computed on a grid of periods and the results can be sensitive to the sampling. As discussed in [1], the performance of the transit periodogram method is more sensitive to the period grid than the `LombScargle` periodogram.

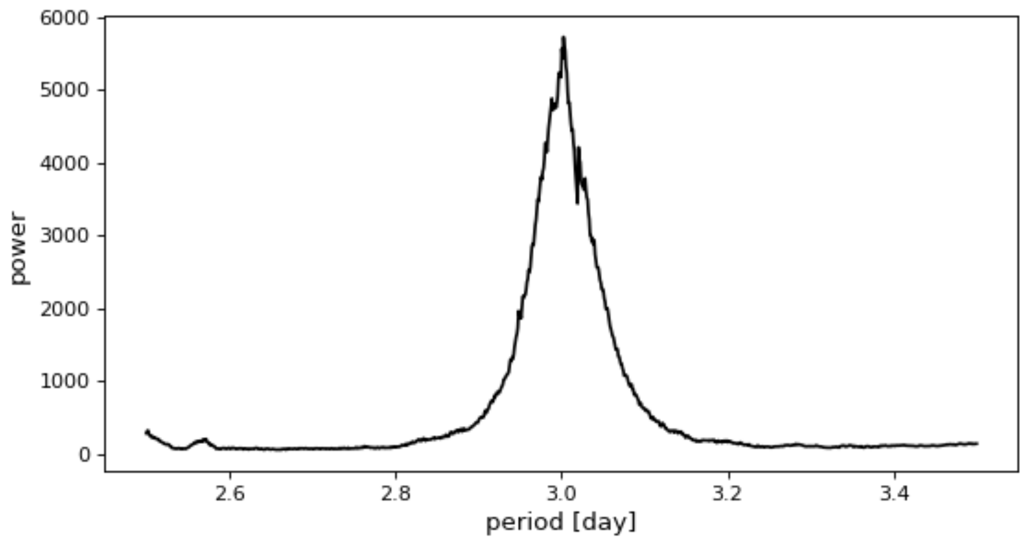
This implementation of the transit periodogram includes a conservative heuristic for estimating the required period grid that is used by the `autoperiod` and `autopower` methods and the details of this method are given in the API documentation for `autoperiod`.

Example

It is possible to provide a specific period grid as follows:

```
>>> model = BoxLeastSquares(t * u.day, y, dy=0.01)
>>> periods = np.linspace(2.5, 3.5, 1000) * u.day
>>> periodogram = model.power(periods, 0.2)
```

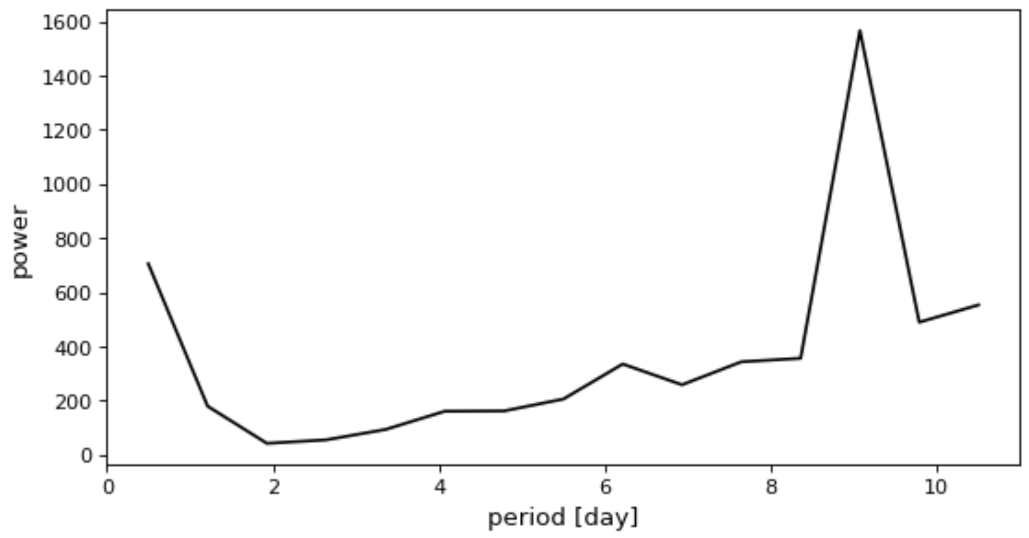
(png, svg, pdf)



However, if the period grid is too coarse, the correct period might be missed.

```
>>> model = BoxLeastSquares(t * u.day, y, dy=0.01)
>>> periods = np.linspace(0.5, 10.5, 15) * u.day
>>> periodogram = model.power(periods, 0.2)
```

(png, svg, pdf)



Peak Statistics

To help in the transit vetting process and to debug problems with candidate peaks, the **compute_stats** method can be used to calculate several statistics of a candidate transit.

Many of these statistics are based on the VARTOOLS package described in [2]. This will often be used as follows to compute stats for the maximum point in the periodogram:

```
>>> model = BoxLeastSquares(t * u.day, y, dy=0.01)
>>> periodogram = model.autopower(0.2)
>>> max_power = np.argmax(periodogram.power)
>>> stats = model.compute_stats(periodogram.period[max_power],
...                             periodogram.duration[max_power],
...                             periodogram.transit_time[max_power])
```

This calculates a dictionary with statistics about this candidate. Each entry in this dictionary is described in the documentation for **compute_stats**.

Literature References

1(1,2,3,4,5) Kovacs, Zucker, & Mazeh (2002), A&A, 391, 369 (arXiv:astro-ph/0206099)

2(1,2) Hartman & Bakos (2016), Astronomy & Computing, 17, 1 (arXiv:1605.06811)

Reference/API

astropy.timeseries Package

This subpackage contains classes and functions for work with time series.

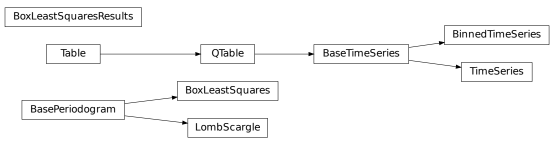
Functions

aggregate_downsample(time_series, *[, …])	Downsample a time series by binning values into bins with a fixed size, using a single function to combine the values in the bin.
autocheck_required_columns(cls)	This is a decorator that ensures that the table contains specific methods indicated by the _required_columns attribute.

Classes

BasePeriodogram(t, y[, dy])	
BaseTimeSeries([data, masked, names, dtype, …])	
BinnedTimeSeries([data, time_bin_start, …])	A class to represent binned time series data in tabular form.
BoxLeastSquares(t, y[, dy])	Compute the box least squares periodogram
BoxLeastSquaresResults(*args)	The results of a BoxLeastSquares search
LombScargle(t, y[, dy, fit_mean, …])	Compute the Lomb-Scargle Periodogram.
TimeSeries([data, time, time_start, …])	A class to represent time series data in tabular form.

Class Inheritance Diagram



astropy.timeseries.io Package

Functions

kepler_fits_reader(filename)	This serves as the FITS reader for KEPLER or TESS files within astropy-timeseries.

Astronomical Coordinate Systems

(astropy.coordinates)

Introduction

The **coordinates** package provides classes for representing a variety of celestial/spatial coordinates and their velocity components, as well as tools for converting between common coordinate systems in a uniform way.

Getting Started

The best way to start using **coordinates** is to use the **SkyCoord** class. **SkyCoord** objects are instantiated by passing in positions (and optional velocities) with specified units and a coordinate frame. Sky positions are commonly passed in as **Quantity** objects and the frame is specified with the string name.

Example

To create a **SkyCoord** object to represent an ICRS (Right ascension [RA], Declination [Dec]) sky position:

```
>>> from astropy import units as u
>>> from astropy.coordinates import SkyCoord
>>> c = SkyCoord(ra=10.625*u.degree, dec=41.2*u.degree, frame='icrs')
```

The initializer for **SkyCoord** is very flexible and supports inputs provided in a number of convenient formats. The following ways of initializing a coordinate are all equivalent to the above:

```
>>> c = SkyCoord(10.625, 41.2, frame='icrs', unit='deg')
>>> c = SkyCoord('00h42m30s', '+41d12m00s', frame='icrs')
>>> c = SkyCoord('00h42.5m', '+41d12m')
>>> c = SkyCoord('00 42 30 +41 12 00', unit=(u.hourangle, u.deg))
>>> c = SkyCoord('00:42.5 +41:12', unit=(u.hourangle, u.deg))
>>> c
<SkyCoord (ICRS): (ra, dec) in deg
    (10.625, 41.2)>
```

The examples above illustrate a few rules to follow when creating a coordinate object:

- Coordinate values can be provided either as unnamed positional arguments or via keyword arguments like `ra` and `dec`, or `l` and `b` (depending on the frame).
- The coordinate `frame` keyword is optional because it defaults to **ICRS**.
- Angle units must be specified for all components, either by passing in a

Quantity object (e.g., `10.5*u.degree`), by including them in the value (e.g., `'+41d12m00s'`), or via the `unit` keyword.

SkyCoord and all other coordinates objects also support array coordinates. These work in the same way as single-value coordinates, but they store multiple coordinates in a single object. When you are going to apply the same operation to many different coordinates (say, from a catalog), this is a better choice than a list of SkyCoord objects, because it will be *much* faster than applying the operation to each SkyCoord in a `for` loop. Like the underlying ndarray instances that contain the data, SkyCoord objects can be sliced, reshaped, etc., and, on `numpy` version 1.17 and later, can be used with functions like numpy.moveaxis, etc., that affect the shape:

```
.. doctest-requires:: numpy>=1.17
```

```
>>> import numpy as np
>>> c = SkyCoord(ra=[10, 11, 12, 13]*u.degree, dec=[41, -5, 42,
0]*u.degree)
>>> c
<SkyCoord (ICRS): (ra, dec) in deg
    [(10., 41.), (11., -5.), (12., 42.), (13.,  0.)]>
>>> c[1]
<SkyCoord (ICRS): (ra, dec) in deg
    (11., -5.)>
>>> c.reshape(2, 2)
<SkyCoord (ICRS): (ra, dec) in deg
    [[(10., 41.), (11., -5.)],
     [(12., 42.), (13.,  0.)]]>
>>> np.roll(c, 1)
<SkyCoord (ICRS): (ra, dec) in deg
    [(13.,  0.), (10., 41.), (11., -5.), (12., 42.)]>
```

Coordinate Access

Once you have a coordinate object you can access the components of that coordinate (e.g., RA, Dec) to get string representations of the full coordinate.

The component values are accessed using (typically lowercase) named attributes that depend on the coordinate frame (e.g., ICRS, Galactic, etc.). For the default, ICRS, the coordinate component names are `ra` and `dec`:

```
>>> c = SkyCoord(ra=10.68458*u.degree, dec=41.26917*u.degree)
>>> c.ra
<Longitude 10.68458 deg>
>>> c.ra.hour
0.7123053333333335
```

```
>>> c.ra.hms
hms_tuple(h=0.0, m=42.0, s=44.299200000000525)
>>> c.dec
<Latitude 41.26917 deg>
>>> c.dec.degree
41.26917
>>> c.dec.radian
0.7202828960652683
```

Coordinates can be converted to strings using the **`to_string()`** method:

```
>>> c = SkyCoord(ra=10.68458*u.degree, dec=41.26917*u.degree)
>>> c.to_string('decimal')
'10.6846 41.2692'
>>> c.to_string('dms')
'10d41m04.488s 41d16m09.012s'
>>> c.to_string('hmsdms')
'00h42m44.2992s +41d16m09.012s'
```

For additional information see the section on Working with Angles.

Transformation

One convenient way to transform to a new coordinate frame is by accessing the appropriately named attribute.

Example

To get the coordinate in the **`Galactic`** frame use:

```
>>> c_icrs = SkyCoord(ra=10.68458*u.degree, dec=41.26917*u.degree,
frame='icrs')
>>> c_icrs.galactic
<SkyCoord (Galactic): (l, b) in deg
    (121.17424181, -21.57288557)>
```

For more control, you can use the **`transform_to`** method, which accepts a frame name, frame class, or frame instance:

```
>>> c_fk5 = c_icrs.transform_to('fk5')  # c_icrs.fk5 does the same
thing
>>> c_fk5
<SkyCoord (FK5: equinox=J2000.000): (ra, dec) in deg
    (10.68459154, 41.26917146)>
```

```
>>> from astropy.coordinates import FK5
>>> c_fk5.transform_to(FK5(equinox='J1975'))  # precess to a
different equinox
<SkyCoord (FK5: equinox=J1975.000): (ra, dec) in deg
    (10.34209135, 41.13232112)>
```

This form of **`transform_to`** also makes it possible to convert from celestial coordinates to **`AltAz`** coordinates, allowing the use of **`SkyCoord`** as a tool for planning observations. For a more complete example of this, see Determining and plotting the altitude/azimuth of a celestial object.

Some coordinate frames such as **`AltAz`** require Earth rotation information (UT1-UTC offset and/or polar motion) when transforming to/from other frames. These Earth rotation values are automatically downloaded from the International Earth Rotation and Reference Systems (IERS) service when required. See IERS data access (astropy.utils.iers) for details of this process.

Representation

So far we have been using a spherical coordinate representation in all of our examples, and this is the default for the built-in frames. Frequently it is convenient to initialize or work with a coordinate using a different representation such as Cartesian or Cylindrical. This can be done by setting the `representation_type` for either **`SkyCoord`** objects or low-level frame coordinate objects.

Example

To initialize or work with a coordinate using a different representation such as Cartesian or Cylindrical:

```
>>> c = SkyCoord(x=1, y=2, z=3, unit='kpc',
representation_type='cartesian')
>>> c
<SkyCoord (ICRS): (x, y, z) in kpc
    (1., 2., 3.)>
>>> c.x, c.y, c.z
(<Quantity 1. kpc>, <Quantity 2. kpc>, <Quantity 3. kpc>)

>>> c.representation_type = 'cylindrical'
>>> c
<SkyCoord (ICRS): (rho, phi, z) in (kpc, deg, kpc)
    (2.23606798, 63.43494882, 3.)>
```

For all of the details see Representations.

Distance

SkyCoord and the individual frame classes also support specifying a distance from the frame origin. The origin depends on the particular coordinate frame; this can be, for example, centered on the earth, centered on the solar system barycenter, etc.

Examples

Two angles and a distance specify a unique point in 3D space, which also allows converting the coordinates to a Cartesian representation:

```
>>> c = SkyCoord(ra=10.68458*u.degree, dec=41.26917*u.degree,
distance=770*u.kpc)
>>> c.cartesian.x
<Quantity 568.71286542 kpc>
>>> c.cartesian.y
<Quantity 107.3008974 kpc>
>>> c.cartesian.z
<Quantity 507.88994292 kpc>
```

With distances assigned, **SkyCoord** convenience methods are more powerful, as they can make use of the 3D information. For example, to compute the physical, 3D separation between two points in space:

```
>>> c1 = SkyCoord(ra=10*u.degree, dec=9*u.degree, distance=10*u.pc,
frame='icrs')
>>> c2 = SkyCoord(ra=11*u.degree, dec=10*u.degree,
distance=11.5*u.pc, frame='icrs')
>>> c1.separation_3d(c2)
<Distance 1.52286024 pc>
```

Convenience Methods

SkyCoord defines a number of convenience methods that support, for example, computing on-sky (i.e., angular) and 3D separations between two coordinates.

Examples

To compute on-sky and 3D separations between two coordinates:

```
>>> c1 = SkyCoord(ra=10*u.degree, dec=9*u.degree, frame='icrs')
>>> c2 = SkyCoord(ra=11*u.degree, dec=10*u.degree, frame='fk5')
>>> c1.separation(c2)  # Differing frames handled correctly
<Angle 1.40453359 deg>
```

Or cross-matching catalog coordinates (detailed in Matching Catalogs):

```
>>> target_c = SkyCoord(ra=10*u.degree, dec=9*u.degree, frame='icrs')
>>> # read in coordinates from a catalog...
>>> catalog_c = ...
>>> idx, sep, _ = target_c.match_to_catalog_sky(catalog_c)
```

The **`astropy.coordinates`** sub-package also provides a quick way to get coordinates for named objects, assuming you have an active internet connection. The **`from_name`** method of **`SkyCoord`** uses Sesame to retrieve coordinates for a particular named object.

To retrieve coordinates for a particular named object:

```
>>> SkyCoord.from_name("PSR J1012+5307")
<SkyCoord (ICRS): (ra, dec) in deg
    (153.1393271, 53.117343)>
```

In some cases, the coordinates are embedded in the catalog name of the object. For such object names, **`from_name`** is able to parse the coordinates from the name if given the `parse=True` option. For slow connections, this may be much faster than a sesame query for the same object name. It's worth noting, however, that the coordinates extracted in this way may differ from the database coordinates by a few deci-arcseconds, so only use this option if you do not need sub-arcsecond accuracy for your coordinates:

```
>>> SkyCoord.from_name("CRTS SSS100805 J194428-420209", parse=True)
<SkyCoord (ICRS): (ra, dec) in deg
    (296.11666667, -42.03583333)>
```

For sites (primarily observatories) on the Earth, **`astropy.coordinates`** provides a quick way to get an **`EarthLocation`** - the **`of_site`** method:

```
>>> from astropy.coordinates import EarthLocation
>>> apo = EarthLocation.of_site('Apache Point Observatory')
>>> apo
<EarthLocation (-1463969.30185172, -5166673.34223433,
3434985.71204565) m>
```

To see the list of site names available, use **astropy.coordinates.EarthLocation.get_site_names()**.

For arbitrary Earth addresses (e.g., not observatory sites), use the **of_address** classmethod. Any address passed to this function uses Google maps to retrieve the latitude and longitude and can also (optionally) query Google maps to get the height of the location. As with Google maps, this works with fully specified addresses, location names, city names, etc.:

```
>>> EarthLocation.of_address('1002 Holy Grail Court, St. Louis, MO')
<EarthLocation (-26726.98216371, -4997009.8604809, 3950271.16507911)
m>
>>> EarthLocation.of_address('1002 Holy Grail Court, St. Louis, MO',
...                          get_height=True)
<EarthLocation (-26727.6272786, -4997130.47437768, 3950367.15622108)
m>
>>> EarthLocation.of_address('Danbury, CT')
<EarthLocation ( 1364606.64511651, -4593292.9428273,
4195415.93695139) m>
```

Note

from_name, **of_site**, and **of_address** are for convenience, and hence are by design relatively low precision. If you need more precise coordinates for an object you should find the appropriate reference and input the coordinates manually, or use more specialized functionality like that in the astroquery or astroplan affiliated packages.

Also note that these methods retrieve data from the internet to determine the celestial or Earth coordinates. The online data may be updated, so if you need to guarantee that your scripts are reproducible in the long term, see the Usage Tips/Suggestions for Methods That Access Remote Resources section.

This functionality can be combined to do more complicated tasks like computing barycentric corrections to radial velocity observations (also a supported high-level **SkyCoord** method - see Radial Velocity Corrections):

```
>>> from astropy.time import Time
>>> obstime = Time('2017-2-14')
>>> target = SkyCoord.from_name('M31')
>>> keck = EarthLocation.of_site('Keck')
>>> target.radial_velocity_correction(obstime=obstime,
location=keck).to('km/s')
<Quantity -22.359784554780255 km / s>
```

Velocities (Proper Motions and Radial Velocities)

In addition to positional coordinates, **coordinates** supports storing and transforming velocities. These are available both via the lower-level coordinate frame classes, and (new in v3.0) via **SkyCoord** objects:

```
>>> sc = SkyCoord(1*u.deg, 2*u.deg, radial_velocity=20*u.km/u.s)
>>> sc
<SkyCoord (ICRS): (ra, dec) in deg
    (1., 2.)
 (radial_velocity) in km / s
    (20.,)>
```

For more details on velocity support (and limitations), see the Working with Velocities in Astropy Coordinates page.

Overview of astropy.coordinates Concepts

Note

The **coordinates** package from v0.4 onward builds from previous versions of the package, and more detailed information and justification of the design is available in APE (Astropy Proposal for Enhancement) 5.

Here we provide an overview of the package and associated framework. This background information is not necessary for using **coordinates**, particularly if you use the **SkyCoord** high-level class, but it is helpful for more advanced usage, particularly creating your own frame, transformations, or representations. Another useful piece of background information are some Important Definitions as they are used in **coordinates**.

coordinates is built on a three-tiered system of objects: representations, frames, and a high-level class. Representations classes are a particular way of storing a three-dimensional data point (or points), such as Cartesian coordinates or spherical polar coordinates. Frames are particular reference frames like FK5 or ICRS, which may store their data in different representations, but have well- defined transformations between each other. These transformations are all stored in the `astropy.coordinates.frame_transform_graph`, and new transformations can be created by users. Finally, the high-level class (**SkyCoord**) uses the frame classes, but provides a more accessible interface to these objects as well as various convenience methods and more string-parsing capabilities.

Separating these concepts makes it easier to extend the functionality of **coordinates**. It allows representations, frames, and transformations to be defined or extended separately, while still preserving the high-level capabilities and ease-of-use of the `SkyCoord` class.

Examples:

See Determining and plotting the altitude/azimuth of a celestial object for an example of using the `coordinates` functionality to prepare for an observing run.

Using `astropy.coordinates`

More detailed information on using the package is provided on separate pages, listed below.

Working with Angles

The angular components of the various coordinate objects are represented by objects of the `Angle` class. While most likely to be encountered in the context of coordinate objects, `Angle` objects can also be used on their own wherever a representation of an angle is needed.

Creation

The creation of an `Angle` object is quite flexible and supports a wide variety of input object types and formats. The type of the input angle(s) can be array, scalar, tuple, string, `Quantity` or another `Angle`. This is best illustrated with a number of examples of valid ways to create an `Angle`.

Examples

There are a number of ways to create an `Angle`:

```
>>> import numpy as np
>>> from astropy import units as u
>>> from astropy.coordinates import Angle

>>> Angle('10.2345d')              # String with 'd' abbreviation for
degrees
<Angle 10.2345 deg>
>>> Angle(['10.2345d', '-20d'])    # Array of strings
<Angle [ 10.2345, -20.    ] deg>
>>> Angle('1:2:30.43 degrees')     # Sexagesimal degrees
<Angle 1.04178611 deg>
>>> Angle('1 2 0 hours')           # Sexagesimal hours
<Angle 1.03333333 hourangle>
```

```
>>> Angle(np.arange(1., 8.), unit=u.deg)  # Numpy array from 1..7 in
degrees
<Angle [1., 2., 3., 4., 5., 6., 7.] deg>
>>> Angle('1°2′3″')               # Unicode degree, arcmin and arcsec
symbols
<Angle 1.03416667 deg>
>>> Angle('1°2′3″N')               # Unicode degree, arcmin, arcsec
symbols and direction
<Angle 1.03416667 deg>
>>> Angle('1d2m3.4s')              # Degree, arcmin, arcsec.
<Angle 1.03427778 deg>
>>> Angle('1d2m3.4sS')              # Degree, arcmin, arcsec,
direction.
<Angle -1.03427778 deg>
>>> Angle('-1h2m3s')               # Hour, minute, second
<Angle -1.03416667 hourangle>
>>> Angle('-1h2m3sW')               # Hour, minute, second, direction
<Angle 1.03416667 hourangle>
>>> Angle((-1, 2, 3), unit=u.deg)  # (degree, arcmin, arcsec)
<Angle -1.03416667 deg>
>>> Angle(10.2345 * u.deg)         # From a Quantity object in
degrees
<Angle 10.2345 deg>
>>> Angle(Angle(10.2345 * u.deg))  # From another Angle object
<Angle 10.2345 deg>
```

Representation

The **`Angle`** object also supports a variety of ways of representing the value of the angle, both as a floating point number and as a string.

Examples

There are many ways to represent the value of an **`Angle`**:

```
>>> a = Angle(1, u.radian)
>>> a
<Angle 1. rad>
>>> a.radian
1.0
>>> a.degree
57.29577951308232
>>> a.hour
3.8197186342054885
>>> a.hms
hms_tuple(h=3.0, m=49.0, s=10.987083139758766)
```

```
>>> a.dms
dms_tuple(d=57.0, m=17.0, s=44.806247096362313)
>>> a.signed_dms
signed_dms_tuple(sign=1.0, d=57.0, m=17.0, s=44.806247096362313)
>>> (-a).dms
dms_tuple(d=-57.0, m=-17.0, s=-44.806247096362313)
>>> (-a).signed_dms
signed_dms_tuple(sign=-1.0, d=57.0, m=17.0, s=44.806247096362313)
>>> a.arcminute
3437.7467707849396
>>> a.to_string()
'1rad'
>>> a.to_string(unit=u.degree)
'57d17m44.8062s'
>>> a.to_string(unit=u.degree, sep=':')
'57:17:44.8062'
>>> a.to_string(unit=u.degree, sep=('deg', 'm', 's'))
'57deg17m44.8062s'
>>> a.to_string(unit=u.hour)
'3h49m10.9871s'
>>> a.to_string(unit=u.hour, decimal=True)
'3.81972'
```

Usage

Angles will also behave correctly for appropriate arithmetic operations.

Example

To use `Angle` objects in arithmetic operations:

```
>>> a = Angle(1.0, u.radian)
>>> a + 0.5 * u.radian + 2 * a
<Angle 3.5 rad>
>>> np.sin(a / 2)
<Quantity 0.47942554>
>>> a == a
array(True, dtype=bool)
>>> a == (a + a)
array(False, dtype=bool)
```

`Angle` objects can also be used for creating coordinate objects.

Example

To create a coordinate object using an `Angle`:

```
>>> from astropy.coordinates import ICRS
>>> ICRS(Angle(1, u.deg), Angle(0.5, u.deg))
<ICRS Coordinate: (ra, dec) in deg
    (1., 0.5)>
```

Wrapping and Bounds

There are two utility methods for working with angles that should have bounds. The **`wrap_at()`** method allows taking an angle or angles and wrapping to be within a single 360 degree slice. The **`is_within_bounds()`** method returns a boolean indicating whether an angle or angles is within the specified bounds.

Longitude and Latitude Objects

`Longitude` and **`Latitude`** are two specialized subclasses of the **`Angle`** class that are used for all of the spherical coordinate classes. **`Longitude`** is used to represent values like right ascension, Galactic longitude, and azimuth (for Equatorial, Galactic, and Alt-Az coordinates, respectively). **`Latitude`** is used for declination, Galactic latitude, and elevation.

Longitude

A **`Longitude`** object is distinguished from a pure **`Angle`** by virtue of a `wrap_angle` property. The `wrap_angle` specifies that all angle values represented by the object will be in the range:

```
wrap_angle - 360 * u.deg <= angle(s) < wrap_angle
```

The default `wrap_angle` is 360 deg. Setting `'wrap_angle=180 * u.deg'` would instead result in values between -180 and +180 deg. Setting the `wrap_angle` attribute of an existing `Longitude` object will result in re-wrapping the angle values in-place. For example:

```
>>> from astropy.coordinates import Longitude
>>> a = Longitude([-20, 150, 350, 360] * u.deg)
>>> a.degree
array([340., 150., 350.,   0.])
>>> a.wrap_angle = 180 * u.deg
>>> a.degree
array([-20., 150., -10.,   0.])
```

Latitude

A Latitude object is distinguished from a pure **Angle** by virtue of being bounded so that:

```
-90.0 * u.deg <= angle(s) <= +90.0 * u.deg
```

Any attempt to set a value outside of that range will result in a **ValueError**.

Using the SkyCoord High-Level Class

The **SkyCoord** class provides a simple and flexible user interface for celestial coordinate representation, manipulation, and transformation between coordinate frames. This is a high-level class that serves as a wrapper around the low-level coordinate frame classes like **ICRS** and **FK5** which do most of the heavy lifting.

The key distinctions between **SkyCoord** and the low-level classes (Using and Designing Coordinate Frames) are as follows:

- The **SkyCoord** object can maintain the union of frame attributes for all built-in and user-defined coordinate frames in the `astropy.coordinates.frame_transform_graph`. Individual frame classes hold only the required attributes (e.g., equinox, observation time, or observer location) for that frame. This means that a transformation from **FK4** (with equinox and observation time) to **ICRS** (with neither) and back to **FK4** via the low-level classes would not remember the original equinox and observation time. Since the **SkyCoord** object stores all attributes, such a round-trip transformation will return to the same coordinate object.
- The **SkyCoord** class is more flexible with inputs to accommodate a wide variety of user preferences and available data formats, whereas the frame classes expect to receive Quantity-like objects with angular units.
- The **SkyCoord** class has a number of convenience methods that are useful in typical analysis.
- At present, **SkyCoord** objects can use only coordinate frames that have transformations defined in the `astropy.coordinates.frame_transform_graph` transform graph object.

Creating SkyCoord Objects

The **SkyCoord** class accepts a wide variety of inputs for initialization. At a minimum, these must provide one or more celestial coordinate values with unambiguous units. Typically you must also specify the coordinate frame, though this is not required.

Common patterns are shown below. In this description the values in upper case like `COORD` or `FRAME` represent inputs which are described in detail in the Initialization Syntax section. Elements in square brackets like `[unit=UNIT]` are optional.

```
SkyCoord(COORD, [FRAME], keyword_args ...)
SkyCoord(LON, LAT, [frame=FRAME], [unit=UNIT], keyword_args ...)
SkyCoord([FRAME], <lon_attr>=LON, <lat_attr>=LAT, keyword_args ...)
```

The examples below illustrate common ways of initializing a **`SkyCoord`** object. These all reflect initializing using spherical coordinates, which is the default for all built-in frames. In order to understand working with coordinates using a different representation, such as Cartesian or cylindrical, see the section on Representations. First, some imports:

```
>>> from astropy.coordinates import SkyCoord  # High-level
coordinates
>>> from astropy.coordinates import ICRS, Galactic, FK4, FK5  # Low-
level frames
>>> from astropy.coordinates import Angle, Latitude, Longitude  #
Angles
>>> import astropy.units as u
>>> import numpy as np
```

Examples

The coordinate values and frame specification can be provided using positional and keyword arguments. First we show positional arguments for RA and Dec:

```
>>> SkyCoord(10, 20, unit='deg')  # Defaults to ICRS
<SkyCoord (ICRS): (ra, dec) in deg
    (10., 20.)>

>>> SkyCoord([1, 2, 3], [-30, 45, 8], frame='icrs', unit='deg')
<SkyCoord (ICRS): (ra, dec) in deg
    [(1., -30.), (2., 45.), (3.,   8.)]>
```

Notice that the first example above does not explicitly give a frame. In this case, the default is taken to be the ICRS system (approximately correct for "J2000" equatorial coordinates). It is always better to explicitly specify the frame when it is known to be ICRS, however, as anyone reading the code will be better able to understand the intent.

String inputs in common formats are acceptable, and the frame can be supplied as either a class type like **FK4**, an instance of a frame class, a **SkyCoord** instance (from which the frame will be extracted), or the lowercase version of a frame name as a string, for example, `"fk4"`:

```
>>> coords = ["1:12:43.2 +1:12:43", "1 12 43.2 +1 12 43"]
>>> sc = SkyCoord(coords, frame=FK4, unit=(u.hourangle, u.deg),
obstime="J1992.21")
>>> sc = SkyCoord(coords, frame=FK4(obstime="J1992.21"),
unit=(u.hourangle, u.deg))
>>> sc = SkyCoord(coords, frame='fk4', unit='hourangle,deg',
obstime="J1992.21")

>>> sc = SkyCoord("1h12m43.2s", "+1d12m43s", frame=Galactic)  # Units
from strings
>>> sc = SkyCoord("1h12m43.2s +1d12m43s", frame=Galactic)  # Units
from string
>>> sc = SkyCoord(l="1h12m43.2s", b="+1d12m43s", frame='galactic')
>>> sc = SkyCoord("1h12.72m +1d12.71m", frame='galactic')
```

Note that frame instances with data and **SkyCoord** instances can only be passed as frames using the `frame=` keyword argument and not as positional arguments.

For representations that have `ra` and `dec` attributes you can supply a coordinate string in a number of other common formats. Examples include:

```
>>> sc = SkyCoord("15h17+89d15")
>>> sc = SkyCoord("275d11m15.6954s+17d59m59.876s")
>>> sc = SkyCoord("8 00 -5 00.6", unit=(u.hour, u.deg))
>>> sc = SkyCoord("J080000.00-050036.00", unit=(u.hour, u.deg))
>>> sc = SkyCoord("J1874221.31+122328.03", unit=u.deg)
```

Astropy **Quantity**-type objects are acceptable and encouraged as a form of input:

```
>>> ra = Longitude([1, 2, 3], unit=u.deg)  # Could also use Angle
>>> dec = np.array([4.5, 5.2, 6.3]) * u.deg  # Astropy Quantity
>>> sc = SkyCoord(ra, dec, frame='icrs')
>>> sc = SkyCoord(ra=ra, dec=dec, frame=ICRS,
obstime='2001-01-02T12:34:56')
```

Finally, it is possible to initialize from a low-level coordinate frame object.

```
>>> c = FK4(1 * u.deg, 2 * u.deg)
>>> sc = SkyCoord(c, obstime='J2010.11', equinox='B1965')  # Override
defaults
```

A key subtlety highlighted here is that when low-level objects are created they have certain default attribute values. For instance, the `FK4` frame uses `equinox='B1950.0` and `obstime=equinox` as defaults. If this object is used to initialize a `SkyCoord` it is possible to override the low-level object attributes that were not explicitly set. If the coordinate above were created with `c = FK4(1 * u.deg, 2 * u.deg, equinox='B1960')` then creating a `SkyCoord` with a different `equinox` would raise an exception.

Initialization Syntax

For spherical representations, which are the most common and are the default input format for all built-in frames, the syntax for `SkyCoord` is given below:

```
SkyCoord(COORD, [FRAME | frame=FRAME], [unit=UNIT], keyword_args ...)
SkyCoord(LON, LAT, [DISTANCE], [FRAME | frame=FRAME], [unit=UNIT],
keyword_args ...)
SkyCoord([FRAME | frame=FRAME], <lon_name>=LON, <lat_name>=LAT,
[unit=UNIT],
         keyword_args ...)
```

In the above description, elements in all capital letters (e.g., `FRAME`) describe a user input of that element type. Elements in square brackets are optional. For nonspherical inputs, see the Representations section.

LON, LAT

Longitude and latitude value can be specified as separate positional arguments. The following options are available for longitude and latitude:

- Single angle value:
  - `Quantity` object
  - Plain numeric value with `unit` keyword specifying the unit
  - Angle string which is formatted for Creation of `Longitude` or `Latitude` objects
- List or `Quantity` array, or NumPy array of angle values
- `Angle`, `Longitude`, or `Latitude` object, which can be scalar or array-valued

Note

While `SkyCoord` is flexible with respect to specifying longitude and latitude component inputs, the frame classes expect to receive `Quantity`-like objects with angular units (i.e., `Angle` or `Quantity`). For example, when specifying components, the frame classes (e.g., `ICRS`) must be created as

```
>>> ICRS(0 * u.deg, 0 * u.deg)
<ICRS Coordinate: (ra, dec) in deg
    (0., 0.)>
```

and other methods of flexible initialization (that work with **SkyCoord**) will not work

```
>>> ICRS(0, 0, unit=u.deg)
UnitTypeError: Longitude instances require units equivalent to
'rad', but no unit was given.
```

DISTANCE

The distance to the object from the frame center can be optionally specified:

- Single distance value:
  - **Quantity** or **Distance** object
  - Plain numeric value for a dimensionless distance
  - Plain numeric value with `unit` keyword specifying the unit
- List, or **Quantity**, or **Distance** array, or NumPy array of angle values

COORD

This input form uses a single object to supply coordinate data. For the case of spherical coordinate frames, the coordinate can include one or more longitude and latitude pairs in one of the following ways:

- Single coordinate string with a LON and LAT value separated by a space. The respective values can be any string which is formatted for Creation of **Longitude** or **Latitude** objects, respectively.
- List or NumPy array of such coordinate strings.
- List of (LON, LAT) tuples, where each LON and LAT are scalars (not arrays).
- `N x 2` NumPy or **Quantity** array of values where the first column is longitude and the second column is latitude, for example, `[[270, -30], [355, +85]] * u.deg`.
- List of (LON, LAT, DISTANCE) tuples.
- `N x 3` NumPy or **Quantity** array of values where columns are longitude, latitude, and distance, respectively.

The input can also be more generalized objects that are not necessarily represented in the standard spherical coordinates:

- Coordinate frame object (e.g., `FK4(1*u.deg, 2*u.deg, obstime='J2012.2')`).
- **SkyCoord** object (which just makes a copy of the object).

- `BaseRepresentation` subclass object like `SphericalRepresentation`, `CylindricalRepresentation`, or `CartesianRepresentation`.

FRAME

This can be a `BaseCoordinateFrame` frame class, an instance of such a class, or the corresponding string alias. The frame classes that are built in to Astropy are `ICRS`, `FK5`, `FK4`, `FK4NoETerms`, `Galactic`, and `AltAz`. The string aliases are lowercase versions of the class name.

If the frame is not supplied then you will see a special `ICRS` identifier. This indicates that the frame is unspecified and operations that require comparing coordinates (even within that object) are not allowed.

unit=UNIT

The unit specifier can be one of the following:

- `Unit` object, which is an angular unit that is equivalent to `Unit('radian')`.
- Single string with a valid angular unit name.
- 2-tuple of `Unit` objects or string unit names specifying the LON and LAT unit, respectively (e.g., `('hourangle', 'degree')`).
- Single string with two unit names separated by a comma (e.g., `'hourangle,degree'`).

If only a single unit is provided then it applies to both LON and LAT.

Other keyword arguments

In lieu of positional arguments to specify the longitude and latitude, the frame-specific names can be used as keyword arguments:

ra, *dec*: **LON**, **LAT** values, optional

RA and Dec for frames where these are representation, including [FIXME] `ICRS`, `FK5`, `FK4`, and `FK4NoETerms`.

l, *b*: **LON**, **LAT** values, optional

Galactic `l` and `b` for the `Galactic` frame.

The following keywords can be specified for any frame:

distance: valid `Distance` initializer, optional

Distance from reference from center to source

obstime: valid `Time` initializer, optional

Time of observation

equinox: valid `Time` initializer, optional

Coordinate frame equinox

If custom user-defined frames are included in the transform graph and they have additional frame attributes, then those attributes can also be set via corresponding keyword arguments in the SkyCoord initialization.

Array Operations

It is possible to store arrays of coordinates in a SkyCoord object, and manipulations done in this way will be orders of magnitude faster than looping over a list of individual SkyCoord objects.

Examples

To store arrays of coordinates in a SkyCoord object:

```
>>> ra = np.linspace(0, 36000, 1001) * u.deg
>>> dec = np.linspace(-90, 90, 1001) * u.deg

>>> sc_list = [SkyCoord(r, d, frame='icrs') for r, d in zip(ra, dec)]
>>> timeit sc_gal_list = [c.galactic for c in sc_list]
1 loops, best of 3: 20.4 s per loop

>>> sc = SkyCoord(ra, dec, frame='icrs')
>>> timeit sc_gal = sc.galactic
100 loops, best of 3: 21.8 ms per loop
```

In addition to vectorized transformations, you can do the usual array slicing, dicing, and selection using the same methods and attributes that you use for ndarray instances. Similarly, on `numpy` version 1.17 or later, corresponding functions as well as others that affect the shape, such as atleast_1d and rollaxis, work as expected. (The relevant functions have to be explicitly enabled in `astropy` source code; let us know if a `numpy` function is not supported that you think should work.):

```
.. doctest-requires:: numpy>=1.17
```

```
>>> north_mask = sc.dec > 0
>>> sc_north = sc[north_mask]
>>> len(sc_north)
500
>>> sc[2:4]
<SkyCoord (ICRS): (ra, dec) in deg
```

```
    [( 72., -89.64), (108., -89.46)]>
>>> sc[500]
<SkyCoord (ICRS): (ra, dec) in deg
    (0., 0.)>
>>> sc[0:-1:100].reshape(2, 5)
<SkyCoord (ICRS): (ra, dec) in deg
    [[(0., -90.), (0., -72.), (0., -54.), (0., -36.), (0., -18.)],
     [(0.,   0.), (0.,  18.), (0.,  36.), (0.,  54.), (0.,  72.)]]>
>>> np.roll(sc[::100], 1)
<SkyCoord (ICRS): (ra, dec) in deg
    [(0.,  90.), (0., -90.), (0., -72.), (0., -54.), (0., -36.),
     (0., -18.), (0.,   0.), (0.,  18.), (0.,  36.), (0.,  54.),
     (0.,  72.)]>
```

Note that similarly to the **ndarray** methods, all but `flatten` try to use new views of the data, with the data copied only if that is impossible (as discussed, for example, in the documentation for NumPy **reshape()**).

Modifying Coordinate Objects In-place

Coordinate values in a array-valued **SkyCoord** object can be modified in-place (added in astropy 4.1). This requires that the new values be set from an another **SkyCoord** object that is equivalent in all ways except for the actual coordinate data values. In this way, no frame transformations are required and the item setting operation is extremely robust.

Specifically, the right hand `value` must be strictly consistent with the object being modified:

- Identical class
- Equivalent frames (**is_equivalent_frame**)
- Identical representation_types
- Identical representation differentials keys
- Identical frame attributes
- Identical “extra” frame attributes (e.g., `obstime` for an ICRS coord)

To modify an array of coordinates in a **SkyCoord** object use the same syntax for a numpy array:

```
>>> sc1 = SkyCoord([1, 2] * u.deg, [3, 4] * u.deg)
>>> sc2 = SkyCoord(10 * u.deg, 20 * u.deg)
>>> sc1[0] = sc2
>>> sc1
<SkyCoord (ICRS): (ra, dec) in deg
    [(10., 20.), ( 2.,  4.)]>
```

You can insert a scalar or array-valued **SkyCoord** object into another compatible **SkyCoord** object:

```
>>> sc1 = SkyCoord([1, 2] * u.deg, [3, 4] * u.deg)
>>> sc2 = SkyCoord(10 * u.deg, 20 * u.deg)
>>> sc1.insert(1, sc2)
<SkyCoord (ICRS): (ra, dec) in deg
    [( 1.,  3.), (10., 20.), ( 2.,  4.)]>
```

With the ability to modify a **SkyCoord** object in-place, all of the Table Operations such as joining, stacking, and inserting are functional with **SkyCoord** mixin columns (so long as no masking is required).

These methods are relatively slow because they require setting from an existing **SkyCoord** object and they perform extensive validation to ensure that the operation is valid. For some applications it may be necessary to take a different lower-level approach which is described in the section Fast In-Place Modification of Coordinates.

Warning

You may be tempted to try an apparently obvious way of modifying a coordinate object in place by updating the component attributes directly, for example `sc1.ra[1] = 40 * u.deg`. However, while this will *appear* to give a correct result it does not actually modify the underlying representation data. This is related to the current implementation of performance-based caching. The current cache implementation is similarly unable to handle in-place changes to the representation (`.data`) or frame attributes such as `.obstime`.

Attributes

The **SkyCoord** object has a number of useful attributes which come in handy. By digging through these we will learn a little bit about **SkyCoord** and how it works.

To begin, one of the most important tools for learning about attributes and methods of objects is “TAB-discovery.” From within IPython you can type an object name, the period, and then the <TAB> key to see what is available. This can often be faster than reading the documentation:

```
>>> sc = SkyCoord(1, 2, frame='icrs', unit='deg', obstime='2013-01-02
14:25:36')
>>> sc.<TAB>
```

```
sc.T                                      sc.match_to_catalog_3d
sc.altaz                                  sc.match_to_catalog_sky
sc.barycentrictrueecliptic                sc.name
sc.cartesian                              sc.ndim
sc.cirs                                   sc.obsgeoloc
sc.copy                                   sc.obsgeovel
sc.data                                   sc.obstime
sc.dec                                    sc.obswl
sc.default_representation                 sc.position_angle
sc.diagonal                               sc.precessedgeocentric
sc.distance                               sc.pressure
sc.equinox                                sc.ra
sc.fk4                                    sc.ravel
sc.fk4noeterms                            sc.realize_frame
sc.fk5                                    sc.relative_humidity
sc.flatten                                sc.represent_as
sc.frame
sc.representation_component_names
sc.frame_attributes
sc.representation_component_units
sc.frame_specific_representation_info     sc.representation_info
sc.from_name                              sc.reshape
sc.from_pixel                             sc.roll
sc.galactic                               sc.search_around_3d
sc.galactocentric                         sc.search_around_sky
sc.galcen_distance                        sc.separation
sc.gcrs                                   sc.separation_3d
sc.geocentrictrueecliptic                 sc.shape
sc.get_constellation                      sc.size
sc.get_frame_attr_names                   sc.skyoffset_frame
sc.guess_from_table                       sc.spherical
sc.has_data                               sc.spherical_offsets_to
sc.hcrs                                   sc.squeeze
sc.heliocentrictrueecliptic               sc.supergalactic
sc.icrs                                   sc.swapaxes
sc.info                                   sc.take
sc.is_equivalent_frame                    sc.temperature
sc.is_frame_attr_default                  sc.to_pixel
sc.is_transformable_to                    sc.to_string
sc.isscalar                               sc.transform_to
sc.itrs                                   sc.transpose
sc.location                               sc.z_sun
```

Here we see many attributes and methods. The most recognizable may be the longitude and latitude attributes which are named `ra` and `dec` for the `ICRS` frame:

```
>>> sc.ra
```

```
<Longitude 1. deg>
>>> sc.dec
<Latitude 2. deg>
```

Next, notice that all of the built-in frame names `icrs`, `galactic`, `fk5`, `fk4`, and `fk4noeterms` are there. Through the magic of Python properties, accessing these attributes calls the object **`transform_to`** method appropriately and returns a new **`SkyCoord`** object in the requested frame:

```
>>> sc_gal = sc.galactic
>>> sc_gal
<SkyCoord (Galactic): (l, b) in deg
    (99.63785528, -58.70969293)>
```

Other attributes you may recognize are `distance`, `equinox`, `obstime`, and `shape`.

Digging Deeper

[Casual users can skip this section]

After transforming to Galactic, the longitude and latitude values are now labeled `l` and `b`, following the normal convention for Galactic coordinates. How does the object know what to call its values? The answer lies in some less obvious attributes:

```
>>> sc_gal.representation_component_names
OrderedDict([('l', 'lon'), ('b', 'lat'), ('distance', 'distance')])

>>> sc_gal.representation_component_units
OrderedDict([('l', Unit("deg")), ('b', Unit("deg"))])

>>> sc_gal.representation_type
<class 'astropy.coordinates.representation.SphericalRepresentation'>
```

Together these tell the object that `l` and `b` are the longitude and latitude, and that they should both be displayed in units of degrees as a spherical-type coordinate (and not, for example, a Cartesian coordinate). Furthermore, the frame's `representation_component_names` attribute defines the coordinate keyword arguments that **`SkyCoord`** will accept.

Another important attribute is `frame_attr_names`, which defines the additional attributes that are required to fully define the frame:

```
>>> sc_fk4 = SkyCoord(1, 2, frame='fk4', unit='deg')
>>> sc_fk4.get_frame_attr_names()
```

```
OrderedDict([('equinox', <Time object: scale='tt' format='byear_str'
value=B1950.000>), ('obstime', None)])
```

The key values correspond to the defaults if no explicit value is provided by the user. This example shows that the **FK4** frame has two attributes, `equinox` and `obstime`, that are required to fully define the frame.

Some trickery is happening here because many of these attributes are actually owned by the underlying coordinate `frame` object which does much of the real work. This is the middle layer in the three-tiered system of objects: representation (spherical, Cartesian, etc.), frame (a.k.a. low-level frame class), and **SkyCoord** (a.k.a. high-level class; see Overview of astropy.coordinates Concepts and Important Definitions):

```
>>> sc.frame
<ICRS Coordinate: (ra, dec) in deg
    (1., 2.)>

>>> sc.has_data is sc.frame.has_data
True

>>> sc.frame.<TAB>
sc.frame.T                                    sc.frame.ra
sc.frame.cartesian                            sc.frame.ravel
sc.frame.copy                                 sc.frame.realize_frame
sc.frame.data                                 sc.frame.represent_as
sc.frame.dec                                  sc.frame.representation
sc.frame.default_representation
sc.frame.representation_component_names
sc.frame.diagonal
sc.frame.representation_component_units
sc.frame.distance
sc.frame.representation_info
sc.frame.flatten                              sc.frame.reshape
sc.frame.frame_attributes                     sc.frame.separation
sc.frame.frame_specific_representation_info  sc.frame.separation_3d
sc.frame.get_frame_attr_names                 sc.frame.shape
sc.frame.has_data                             sc.frame.size
sc.frame.is_equivalent_frame                  sc.frame.spherical
sc.frame.is_frame_attr_default                sc.frame.squeeze
sc.frame.is_transformable_to                  sc.frame.swapaxes
sc.frame.isscalar                             sc.frame.take
sc.frame.name                                 sc.frame.transform_to
sc.frame.ndim                                 sc.frame.transpose

>>> sc.frame.name
'icrs'
```

The `SkyCoord` object exposes the `frame` object attributes as its own. Though it might seem a tad confusing at first, this is a good thing because it makes `SkyCoord` objects and `BaseCoordinateFrame` objects behave very similarly and most routines can accept either one as input without much bother (duck typing!).

The lowest layer in the stack is the abstract `UnitSphericalRepresentation` object:

```
>>> sc_gal.frame.data
<UnitSphericalRepresentation (lon, lat) in rad
    (1.73900863, -1.02467744)>
```

Transformations

The topic of transformations is covered in detail in the section on Transforming between Systems.

For completeness, here we will give some examples. Once you have defined your coordinates and the reference frame, you can transform from that frame to another frame. You can do this in a few different ways: if you only want the default version of that frame, you can use attribute-style access (as mentioned previously). For more control, you can use the `transform_to` method, which accepts a frame name, frame class, frame instance, or `SkyCoord`.

Examples
To transform from one frame to another:

```
>>> from astropy.coordinates import FK5
>>> sc = SkyCoord(1, 2, frame='icrs', unit='deg')
>>> sc.galactic
<SkyCoord (Galactic): (l, b) in deg
    (99.63785528, -58.70969293)>

>>> sc.transform_to('fk5')  # Same as sc.fk5 and sc.transform_to(FK5)
<SkyCoord (FK5: equinox=J2000.000): (ra, dec) in deg
        (1.00000656, 2.00000243)>

>>> sc.transform_to(FK5(equinox='J1975'))  # Transform to FK5 with a
different equinox
<SkyCoord (FK5: equinox=J1975.000): (ra, dec) in deg
        (0.67967282, 1.86083014)>
```

Transforming to a `SkyCoord` instance is a convenient way of ensuring that two coordinates are in the exact same reference frame:

```
>>> sc2 = SkyCoord(3, 4, frame='fk4', unit='deg',
obstime='J1978.123', equinox='B1960.0')
>>> sc.transform_to(sc2)
<SkyCoord (FK4: equinox=B1960.000, obstime=J1978.123): (ra, dec) in
deg
    (0.48726331, 1.77731617)>
```

Representations

So far we have been using a spherical coordinate representation in all of the examples, and this is the default for the built-in frames. Frequently it is convenient to initialize or work with a coordinate using a different representation such as Cartesian or cylindrical. In this section, we discuss how to initialize an object using a different representation and how to change the representation of an object. For more information about representation objects themselves, see Using and Designing Coordinate Representations.

Initialization

Most of what you need to know can be inferred from the examples below and by extrapolating the previous documentation for spherical representations. Initialization requires setting the `representation_type` keyword and supplying the corresponding components for that representation.

Examples

To initialize an object using a representation type other than spherical:

```
>>> c = SkyCoord(x=1, y=2, z=3, unit='kpc',
representation_type='cartesian')
>>> c
<SkyCoord (ICRS): (x, y, z) in kpc
    (1., 2., 3.)>
>>> c.x, c.y, c.z
(<Quantity 1. kpc>, <Quantity 2. kpc>, <Quantity 3. kpc>)
```

Other variations include:

```
>>> SkyCoord(1, 2*u.deg, 3, representation_type='cylindrical')
<SkyCoord (ICRS): (rho, phi, z) in (, deg, )
    (1., 2., 3.)>

>>> SkyCoord(rho=1*u.km, phi=2*u.deg, z=3*u.m,
representation_type='cylindrical')
```

```
<SkyCoord (ICRS): (rho, phi, z) in (km, deg, m)
    (1., 2., 3.)>

>>> SkyCoord(rho=1, phi=2, z=3, unit=(u.km, u.deg, u.m),
representation_type='cylindrical')
<SkyCoord (ICRS): (rho, phi, z) in (km, deg, m)
    (1., 2., 3.)>

>>> SkyCoord(1, 2, 3, unit=(None, u.deg, None),
representation_type='cylindrical')
<SkyCoord (ICRS): (rho, phi, z) in (, deg, )
    (1., 2., 3.)>
```

In general terms, the allowed syntax is as follows:

```
SkyCoord(COORD, [FRAME | frame=FRAME], [unit=UNIT],
[representation_type=REPRESENTATION],
         keyword_args ...)
SkyCoord(COMP1, COMP2, [COMP3], [FRAME | frame=FRAME], [unit=UNIT],
         [representation_type=REPRESENTATION], keyword_args ...)
SkyCoord([FRAME | frame=FRAME], <comp1_name>=COMP1,
<comp2_name>=COMP2,
         <comp3_name>=COMP3, [representation_type=REPRESENTATION],
[unit=UNIT],
         keyword_args ...)
```

In this case, the `keyword_args` now includes the element `representation_type=REPRESENTATION`. In the above description, elements in all capital letters (e.g., `FRAME`) describe a user input of that element type. Elements in square brackets are optional.

COMP1, **COMP2**, **COMP3**

Component values can be specified as separate positional arguments or as keyword arguments. In this formalism the exact type of allowed input depends on the details of the representation. In general, the following input forms are supported:

- Single value:
  - Component class object
  - Plain numeric value with `unit` keyword specifying the unit
- List or component class array, or NumPy array of values

Each representation component has a specified class (the “component class”) which is used to convert generic input data into a predefined object class with a certain unit. These component classes are expected to be subclasses of the **`Quantity`** class.

COORD

This input form uses a single object to supply coordinate data. The coordinate can specify one or more coordinate positions as follows:

- List of `(COMP1, .., COMP<M>)` tuples, where each component is a scalar (not array) and there are `M` components in the representation. Typically there are three components, but some (e.g., **`UnitSphericalRepresentation`**) can have fewer.
- `N x M` NumPy or **`Quantity`** array of values, where `N` is the number of coordinates and `M` is the number of components.

REPRESENTATION

The representation can be supplied either as a **`BaseRepresentation`** class (e.g., **`CartesianRepresentation`**) or as a string name that is simply the class name in lowercase without the `'representation'` suffix (e.g., `'cartesian'`).

The rest of the inputs for creating a **`SkyCoord`** object in the general case are the same as for spherical.

Details

The available set of representations is dynamic and may change depending on what representation classes have been defined. The built-in representations are:

Name	Class
`spherical`	`SphericalRepresentation`
`unitspherical`	`UnitSphericalRepresentation`
`physicsspherical`	`PhysicsSphericalRepresentation`
`cartesian`	`CartesianRepresentation`
`cylindrical`	`CylindricalRepresentation`

Each frame knows about all of the available representations, but different frames may use different names for the same components. A common example is that the **`Galactic`** frame uses `l` and `b` instead of `ra` and `dec` for the `lon` and `lat` components of the **`SphericalRepresentation`**.

For a particular frame, in order to see the full list of representations and how it names all of the components, first make an instance of that frame without any data, and then print the `representation_info` property:

```
>>> ICRS().representation_info
{astropy.coordinates.representation.CartesianRepresentation:
  {'names': ('x', 'y', 'z'),
   'units': (None, None, None)},
 astropy.coordinates.representation.SphericalRepresentation:
  {'names': ('ra', 'dec', 'distance'),
   'units': (Unit("deg"), Unit("deg"), None)},
 astropy.coordinates.representation.UnitSphericalRepresentation:
  {'names': ('ra', 'dec'),
   'units': (Unit("deg"), Unit("deg"))},
 astropy.coordinates.representation.PhysicsSphericalRepresentation:
  {'names': ('phi', 'theta', 'r'),
   'units': (Unit("deg"), Unit("deg"), None)},
 astropy.coordinates.representation.CylindricalRepresentation:
  {'names': ('rho', 'phi', 'z'),
   'units': (None, Unit("deg"), None)}
}
```

This is a bit messy but it shows that for each representation there is a `dict` with two keys:

- `names`: defines how each component is named in that frame.
- `units`: defines the units of each component when output, where `None` means to not force a particular unit.

For a particular coordinate instance you can use the `representation_type` attribute in conjunction with the `representation_component_names` attribute to figure out what keywords are accepted by a particular class object. The former will be the representation class the system is expressed in (e.g., spherical for equatorial frames), and the latter will be a dictionary mapping names for that frame to the component name on the representation class:

```
>>> import astropy.units as u
>>> icrs = ICRS(1*u.deg, 2*u.deg)
>>> icrs.representation_type
<class 'astropy.coordinates.representation.SphericalRepresentation'>
>>> icrs.representation_component_names
OrderedDict([('ra', 'lon'), ('dec', 'lat'), ('distance',
'distance')])
```

Changing Representation

The representation of the coordinate object can be changed, as shown below. This actually does *nothing* to the object internal data which stores the coordinate values, but it changes the external view of that data in two ways:

- The object prints itself in accord with the new representation.
- The available attributes change to match those of the new representation (e.g., from `ra, dec, distance` to `x, y, z`).

Setting the `representation_type` thus changes a *property* of the object (how it appears) without changing the intrinsic object itself which represents a point in 3D space.

Examples

To change the representation of a coordinate object by setting the `representation_type`

```
>>> c = SkyCoord(x=1, y=2, z=3, unit='kpc',
representation_type='cartesian')
>>> c
<SkyCoord (ICRS): (x, y, z) in kpc
    (1., 2., 3.)>

>>> c.representation_type = 'cylindrical'
>>> c
<SkyCoord (ICRS): (rho, phi, z) in (kpc, deg, kpc)
    (2.23606798, 63.43494882, 3.)>
>>> c.phi.to(u.deg)
<Angle 63.43494882 deg>
>>> c.x
Traceback (most recent call last):
...
AttributeError: 'SkyCoord' object has no attribute 'x'

>>> c.representation_type = 'spherical'
>>> c
<SkyCoord (ICRS): (ra, dec, distance) in (deg, deg, kpc)
    (63.43494882, 53.3007748, 3.74165739)>

>>> c.representation_type = 'unitspherical'
>>> c
<SkyCoord (ICRS): (ra, dec) in deg
    (63.43494882, 53.3007748)>
```

You can also use any representation class to set the representation:

```
>>> from astropy.coordinates import CartesianRepresentation
>>> c.representation_type = CartesianRepresentation
```

Note that if all you want is a particular representation without changing the state of the **SkyCoord** object, you should instead use the `astropy.coordinates.SkyCoord.represent_as()` method:

```
>>> c.representation_type = 'spherical'
>>> cart = c.represent_as(CartesianRepresentation)
>>> cart
```

```
<CartesianRepresentation (x, y, z) in kpc
    (1., 2., 3.)>
>>> c.representation_type
<class 'astropy.coordinates.representation.SphericalRepresentation'>
```

Example 1: Plotting random data in Aitoff projection

This is an example of how to make a plot in the Aitoff projection using data in a `SkyCoord` object. Here, a randomly generated data set will be used.

First we need to import the required packages. We use matplotlib here for plotting and numpy to get the value of pi and to generate our random data.

```
>>> from astropy import units as u
>>> from astropy.coordinates import SkyCoord
>>> import numpy as np
```

We now generate random data for visualization. For RA this is done in the range of 0 and 360 degrees (`ra_random`), for DEC between -90 and +90 degrees (`dec_random`). Finally, we multiply these values by degrees to get a `Quantity` with units of degrees.

```
>>> ra_random = np.random.rand(100)*360.0 * u.degree
>>> dec_random = (np.random.rand(100)*180.0-90.0) * u.degree
```

As the next step, those coordinates are transformed into an `astropy.coordinates SkyCoord` object.

```
>>> c = SkyCoord(ra=ra_random, dec=dec_random, frame='icrs')
```

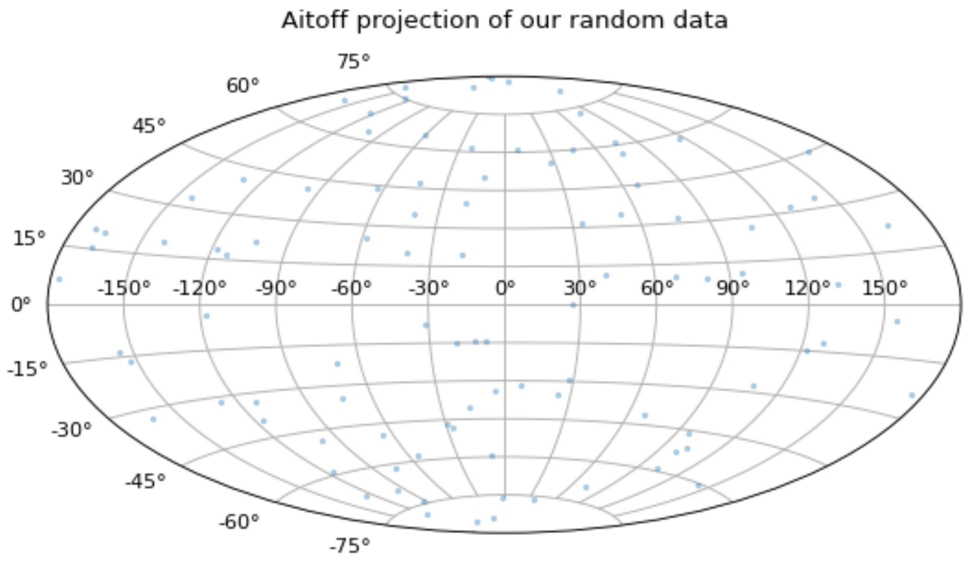
Because matplotlib needs the coordinates in radians and between \(-\pi\) and \(\pi\), not 0 and \(2\pi\), we have to convert them. For this purpose the `astropy.coordinates.Angle` object provides a special method, which we use here to wrap at 180:

```
>>> ra_rad = c.ra.wrap_at(180 * u.deg).radian
>>> dec_rad = c.dec.radian
```

As a last step, we set up the plotting environment with matplotlib using the Aitoff projection with a specific title, a grid, filled circles as markers with a marker size of 2, and an alpha value of 0.3. We use a figure with an x-y ratio that is well suited for such a projection and we move the title upwards from its usual position to avoid overlap with the axis labels.

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(8,4.2))
>>> plt.subplot(111, projection="aitoff")
>>> plt.title("Aitoff projection of our random data")
>>> plt.grid(True)
>>> plt.plot(ra_rad, dec_rad, 'o', markersize=2, alpha=0.3)
>>> plt.subplots_adjust(top=0.95,bottom=0.0)
>>> plt.show()
```

(png, svg, pdf)



Example 2: Plotting star positions in bulge and disk

This is a more realistic example of how to make a plot in the Aitoff projection using data in a **SkyCoord** object. Here, a randomly generated data set (multivariate normal distribution) for both stars in the bulge and in the disk of a galaxy will be used. Both types will be plotted with different number counts.

As in the last example, we first import the required packages.

```
>>> from astropy import units as u
>>> from astropy.coordinates import SkyCoord
>>> import numpy as np
```

We now generate random data for visualization using **numpy.random.Generator.multivariate_normal**.

```
>>> disk = np.random.multivariate_normal(mean=[0,0,0],
cov=np.diag([1,1,0.5]), size=5000)
>>> bulge = np.random.multivariate_normal(mean=[0,0,0],
cov=np.diag([1,1,1]), size=500)
>>> galaxy = np.concatenate([disk, bulge])
```

As the next step, those coordinates are transformed into an **astropy.coordinates SkyCoord** object.

```
>>> c_gal = SkyCoord(galaxy, representation_type='cartesian',
frame='galactic')
>>> c_gal_icrs = c_gal.icrs
```

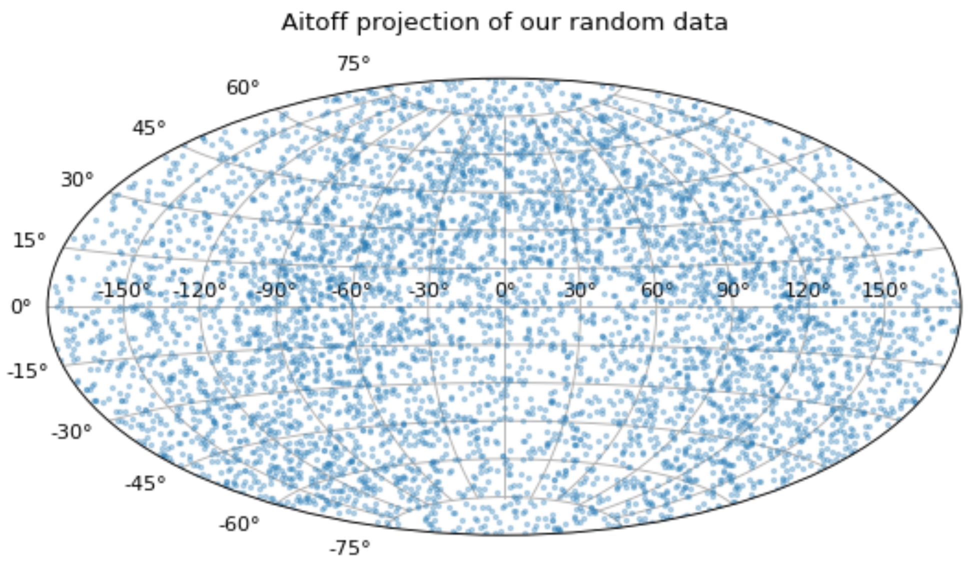
Again, as in the last example, we need to convert the coordinates in radians and make sure they are between \(-\pi\) and \(\pi\):

```
>>> ra_rad = c_gal_icrs.ra.wrap_at(180 * u.deg).radian
>>> dec_rad = c_gal_icrs.dec.radian
```

We use the same plotting setup as in the last example:

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(figsize=(8,4.2))
>>> plt.subplot(111, projection="aitoff")
>>> plt.title("Aitoff projection of our random data")
>>> plt.grid(True)
>>> plt.plot(ra_rad, dec_rad, 'o', markersize=2, alpha=0.3)
>>> plt.subplots_adjust(top=0.95,bottom=0.0)
>>> plt.show()
```

(png, svg, pdf)



Comparing SkyCoord Objects

There are two primary ways to compare **SkyCoord** objects to each other. First is checking if the coordinates are within a specified distance of each other. This is what most users should do in their science or processing analysis work because it allows for a tolerance due to floating point representation issues. The second is checking for exact equivalence of two objects down to the bit, which is most useful for developers writing tests.

The example below illustrates the floating point issue using the exact equality comparison, where we do a roundtrip transformation FK4 => ICRS => FK4 and then compare:

```
>>> sc1 = SkyCoord(1*u.deg, 2*u.deg, frame='fk4')
>>> sc1.icrs.fk4 == sc1
False
```

Matching Within Tolerance

To test if coordinates are within a certain angular distance of one other, use the **separation** method:

```
>>> sc1.icrs.fk4.separation(sc1).to(u.arcsec)
<Angle 7.98873629e-13 arcsec>
>>> sc1.icrs.fk4.separation(sc1) < 1e-9 * u.arcsec
True
```

Exact Equality

Astropy also provides an exact equality operator for coordinates. For example, when comparing, e.g., two **SkyCoord** objects:

```
>>> left_skycoord == right_skycoord
```

the right object must be strictly consistent with the left object for comparison:

- Identical class
- Equivalent frames (**is_equivalent_frame**)
- Identical representation_types
- Identical representation differentials keys
- Identical frame attributes
- Identical "extra" frame attributes (e.g., `obstime` for an ICRS coord)

In the first example we show simple comparisons using array-valued coordinates:

```
>>> sc1 = SkyCoord([1, 2]*u.deg, [3, 4]*u.deg)
>>> sc2 = SkyCoord([1, 20]*u.deg, [3, 4]*u.deg)
>>> sc1 == sc2  # Array-valued comparison
array([ True, False])
>>> sc2 == sc2[1]  # Broadcasting comparison with a scalar
array([False,  True])
>>> sc2[0] == sc2[1]  # Scalar to scalar comparison
False
>>> sc1 != sc2  # Not equal
array([False,  True])
```

In addition to numerically comparing the representation component data (which may include velocities), the equality comparison includes strict tests that all of the frame attributes like `equinox` or `obstime` are exactly equal. Any mismatch in attributes will result in an exception being raised. For example:

```
>>> sc1 = SkyCoord([1, 2]*u.deg, [3, 4]*u.deg)
>>> sc2 = SkyCoord([1, 20]*u.deg, [3, 4]*u.deg, obstime='2020-01-01')
>>> sc1 == sc2
...
ValueError: cannot compare: extra frame attribute 'obstime' is not
equivalent
 (perhaps compare the frames directly to avoid this exception)
```

In this example the `obstime` attribute is a so-called “extra” frame attribute that does not apply directly to the ICRS coordinate frame. So we could compare with the following, this time using the `!=` operator for variety:

```
>>> sc1.frame != sc2.frame
array([False, True])
```

One slightly special case is comparing two frames that both have no data, where the return value is the same as `frame1.is_equivalent_frame(frame2)`. For example:

```
>>> from astropy.coordinates import FK4
>>> FK4() == FK4(obstime='2020-01-01')
False
```

Convenience Methods

A number of convenience methods are available, and you are encouraged to read the available docstrings below:

- **match_to_catalog_sky**,

- `match_to_catalog_3d`,
- `position_angle`,
- `separation`,
- `separation_3d`
- `apply_space_motion`

Additional information and examples can be found in the section on Separations, Offsets, Catalog Matching, and Related Functionality and Accounting for Space Motion.

Transforming between Systems

`astropy.coordinates` supports a rich system for transforming coordinates from one frame to another. While common astronomy frames are built into Astropy, the transformation infrastructure is dynamic. This means it allows users to define new coordinate frames and their transformations. The topic of writing your own coordinate frame or transforms is detailed in Defining a New Frame, and this section is focused on how to *use* transformations.

The full list of built-in coordinate frames, the included transformations, and the frame names are shown as a (clickable) graph in the `coordinates` API documentation.

Examples

The recommended method of transformation is shown below:

```
>>> import astropy.units as u
>>> from astropy.coordinates import SkyCoord
>>> gc = SkyCoord(l=0*u.degree, b=45*u.degree, frame='galactic')
>>> gc.fk5
<SkyCoord (FK5: equinox=J2000.000): (ra, dec) in deg
    ( 229.27251463, -1.12844288)>
```

While this appears to be ordinary attribute-style access, it is actually syntactic sugar for the more general `transform_to()` method, which can accept either a frame name, class, or instance:

```
>>> from astropy.coordinates import FK5
>>> gc.transform_to('fk5')
<SkyCoord (FK5: equinox=J2000.000): (ra, dec) in deg
    ( 229.27251463, -1.12844288)>
>>> gc.transform_to(FK5)
<SkyCoord (FK5: equinox=J2000.000): (ra, dec) in deg
```

```
    ( 229.27251463, -1.12844288)>
>>> gc.transform_to(FK5(equinox='J1980.0'))
<SkyCoord (FK5: equinox=J1980.000): (ra, dec) in deg
    ( 229.0146935, -1.05560349)>
```

As a convenience, it is also possible to use a **SkyCoord** object as the frame in **transform_to()**. This allows for putting one coordinate object into the frame of another:

```
>>> sc = SkyCoord(ra=1.0, dec=2.0, unit='deg', frame=FK5,
equinox='J1980.0')
>>> gc.transform_to(sc)
<SkyCoord (FK5: equinox=J1980.000): (ra, dec) in deg
    ( 229.0146935, -1.05560349)>
```

Some coordinate frames (including **FK5**, **FK4**, and **FK4NoETerms**) support “self transformations,” meaning the *type* of frame does not change, but the frame attributes do. Any example is precessing a coordinate from one equinox to another in an equatorial frame. This is done by passing `transform_to` a frame class with the relevant attributes, as shown below. Note that these frames use a default equinox if you do not specify one:

```
>>> fk5c = SkyCoord('02h31m49.09s', '+89d15m50.8s', frame=FK5)
>>> fk5c.equinox
<Time object: scale='tt' format='jyear_str' value=J2000.000>
>>> fk5c
<SkyCoord (FK5: equinox=J2000.000): (ra, dec) in deg
    ( 37.95454167,  89.26411111)>
>>> fk5_2005 = FK5(equinox='J2005')  # String initializes an
astropy.time.Time object
>>> fk5c.transform_to(fk5_2005)
<SkyCoord (FK5: equinox=J2005.000): (ra, dec) in deg
    ( 39.39317639,  89.28584422)>
```

You can also specify the equinox when you create a coordinate using a **Time** object:

```
>>> from astropy.time import Time
>>> fk5c = SkyCoord('02h31m49.09s', '+89d15m50.8s',
...                 frame=FK5(equinox=Time('J1970')))
>>> fk5_2000 = FK5(equinox=Time(2000, format='jyear'))
>>> fk5c.transform_to(fk5_2000)
<SkyCoord (FK5: equinox=2000.0): (ra, dec) in deg
    ( 48.023171,  89.38672485)>
```

The same lower-level frame classes also have a `transform_to()` method that works the same as above, but they do not support attribute-style access. They are also subtly different in that they only use frame attributes present in the initial or final frame, while `SkyCoord` objects use any frame attributes they have for all transformation steps. So `SkyCoord` can always transform from one frame to another and back again without change, while low-level classes may lose information and hence often do not round-trip.

Transformations and Solar System Ephemerides

Some transformations (e.g., the transformation between `ICRS` and `GCRS`) require the use of a Solar System ephemeris to calculate the position and velocity of the Earth and Sun. By default, transformations are calculated using built-in ERFA routines, but they can also use more precise ones using the JPL ephemerides (which are derived from dynamical models).

Example

To use the JPL ephemerides, use the `solar_system_ephemeris` context manager, as shown below:

```
>>> from astropy.coordinates import solar_system_ephemeris
>>> from astropy.coordinates import GCRS
>>> with solar_system_ephemeris.set('jpl'):
...     fk5c.transform_to(GCRS(obstime=Time("J2000")))
```

For locations at large distances from the Solar system, using the JPL ephemerides will make a negligible difference on the order of micro-arcseconds. For nearby objects, such as the Moon, the difference can be of the order of milli-arcseconds. For more details about what ephemerides are available, including the requirements for using JPL ephemerides, see Solar System Ephemerides.

Solar System Ephemerides

`astropy.coordinates` can calculate the `SkyCoord` of some of the major solar system objects. By default, it uses approximate orbital elements calculated using PyERFA routines, but it can also use more precise ones using the JPL ephemerides (which are derived from dynamical models). The default JPL ephemerides (DE430) provide predictions valid roughly for the years between 1550 and 2650. The file is 115 MB and will need to be downloaded the first time you use this functionality, but will be cached after that.

Note

Using JPL ephemerides requires that the jplephem package be installed. This is most conveniently achieved via `pip install jplephem`, although whatever package management system you use might have it as well.

Three functions are provided; **`get_body()`**, **`get_moon()`** and **`get_body_barycentric()`**. The first two functions return **`SkyCoord`** objects in the **`GCRS`** frame, while the latter returns a **`CartesianRepresentation`** of the barycentric position of a body (i.e., in the **`ICRS`** frame).

Examples

Here is an example of using these functions with built-in ephemerides (i.e., without the need to download a large ephemerides file):

```
>>> from astropy.time import Time
>>> from astropy.coordinates import solar_system_ephemeris,
EarthLocation
>>> from astropy.coordinates import get_body_barycentric, get_body,
get_moon
>>> t = Time("2014-09-22 23:22")
>>> loc = EarthLocation.of_site('greenwich')
>>> with solar_system_ephemeris.set('builtin'):
...     jup = get_body('jupiter', t, loc)
>>> jup
<SkyCoord (GCRS: obstime=2014-09-22 23:22:00.000, obsgeoloc=
(3949481.68990863, -550931.91188162, 4961151.73733451) m, obsgeovel=
(40.15954083, 287.47876693, -0.04597867) m / s): (ra, dec, distance)
in (deg, deg, AU)
    (136.91116209, 17.02935409, 5.94386022)>
```

Above, we used `solar_system_ephemeris` as a context, which sets the default ephemeris while in the `with` clause, and resets it at the end.

To get more precise positions than is possible with the built-in ephemeris (see Precision of the Built-In Ephemeris), you could use the `de430` ephemeris mentioned above, or, if you only care about times between 1950 and 2050, opt for the `de432s` ephemeris, which is stored in a smaller, ~10 MB, file (which will be downloaded and cached when the ephemeris is set):

```
>>> solar_system_ephemeris.set('de432s')
<ScienceState solar_system_ephemeris: 'de432s'>
>>> get_body('jupiter', t, loc)
<SkyCoord (GCRS: obstime=2014-09-22 23:22:00.000, obsgeoloc=
(3949481.69230491, -550931.90674055, 4961151.73597586) m, obsgeovel=
(40.15954083, 287.47863521, -0.0459789) m / s): (ra, dec, distance)
in (deg, deg, km)
    (136.90234802, 17.03160667, 8.89196021e+08)>
>>> get_moon(t, loc)
<SkyCoord (GCRS: obstime=2014-09-22 23:22:00.000, obsgeoloc=
(3949481.69230491, -550931.90674055, 4961151.73597586) m, obsgeovel=
(40.15954083, 287.47863521, -0.0459789) m / s): (ra, dec, distance)
in (deg, deg, km)
    (165.51849203, 2.32863886, 407229.6503193)>
>>> get_body_barycentric('moon', t)
<CartesianRepresentation (x, y, z) in km
    (  1.50107535e+08, -866789.11996916, -418963.55218495)>
```

For one-off calculations with a given ephemeris, you can also pass it directly to the various functions:

```
>>> get_body_barycentric('moon', t, ephemeris='de432s')
...
<CartesianRepresentation (x, y, z) in km
    (  1.50107535e+08, -866789.11996916, -418963.55218495)>
>>> get_body_barycentric('moon', t, ephemeris='builtin')
...
<CartesianRepresentation (x, y, z) in km
    (  1.50107516e+08, -866828.92702829, -418980.15907332)>
```

For a list of the bodies for which positions can be calculated, do:

```
>>> solar_system_ephemeris.bodies
('sun',
 'mercury',
 'venus',
 'earth-moon-barycenter',
 'earth',
 'moon',
 'mars',
 'jupiter',
 'saturn',
 'uranus',
 'neptune',
 'pluto')
>>> solar_system_ephemeris.set('builtin')
<ScienceState solar_system_ephemeris: 'builtin'>
>>> solar_system_ephemeris.bodies
('earth',
 'sun',
 'moon',
 'mercury',
 'venus',
 'earth-moon-barycenter',
 'mars',
 'jupiter',
 'saturn',
 'uranus',
 'neptune')
```

Note

While the sun is included in the these ephemerides, it is important to recognize that **get_sun** always uses the built-in, polynomial model (as this requires no special download). So it is not safe to assume that `get_body(time, 'sun')` and `get_sun(time)` will give the same result.

Precision of the Built-In Ephemeris

The algorithm for calculating positions and velocities for planets other than Earth used by ERFA is due to J.L. Simon, P. Bretagnon, J. Chapront, M. Chapront-Touze, G. Francou and J. Laskar (Bureau des Longitudes, Paris, France). From comparisons with JPL ephemeris DE102, they quote the maximum errors over the interval 1800-2050 below. For more details, see the PyERFA routine, **erfa.plan94**. For the Earth, the rms errors in position and velocity are about 4.6 km and 1.4 mm/s, respectively (see **erfa.epv00**).

Planet	L (arcsec)	B (arcsec)	R (km)
Mercury	4	1	300
Venus	5	1	800
EMB	6	1	1000
Mars	17	1	7700
Jupiter	71	5	76000
Saturn	81	13	267000
Uranus	86	7	712000
Neptune	11	1	253000

Working with Earth Satellites Using Astropy Coordinates

Satellite data is normally provided in the Two-Line Element (TLE) format (see here for a definition). These datasets are designed to be used in combination with a theory for orbital propagation model to predict the positions of satellites.

The history of such models is discussed in detail in Vallado et al (2006) who also provide a reference implementation of the SGP4 orbital propagation code, designed to be compatible with the TLE sets provided by the United States

Department of Defense, which are available from a source like Celestrak.

The output coordinate frame of the SGP4 model is the True Equator, Mean Equinox frame (TEME), which is one of the frames built-in to **`astropy.coordinates`**. TEME is an Earth-centered inertial frame (i.e., it does not rotate with respect to the stars). Several definitions exist; `astropy` uses the implementation described in Vallado et al (2006).

Finding TEME Coordinates from TLE Data

There is currently no support in **`astropy.coordinates`** for computing satellite orbits from TLE orbital element sets. Full support for handling TLE files is available in the Skyfield library, but some advice for dealing with satellite data in `astropy` is below.

You will need some external library to compute the position and velocity of the satellite from the TLE orbital elements. The SGP4 library can do this. An example of using this library to find the **`TEME`** coordinates of a satellite is:

```
>>> from sgp4.api import Satrec
>>> from sgp4.api import SGP4_ERRORS
>>> s = '1 25544U 98067A   19343.69339541  .00001764  00000-0
38792-4 0  9991'
>>> t = '2 25544  51.6439 211.2001 0007417  17.6667  85.6398
15.50103472202482'
>>> satellite = Satrec.twoline2rv(s, t)
```

The `satellite` object has a method, `satellite.sgp4`, that will try to compute the TEME position and velocity at a given time:

```
>>> from astropy.time import Time
>>> t = Time(2458827.362605, format='jd')
>>> error_code, teme_p, teme_v = satellite.sgp4(t.jd1, t.jd2)  # in
km and km/s
>>> if error_code != 0:
...     raise RuntimeError(SGP4_ERRORS[error_code])
```

Now that we have the position and velocity in kilometers and kilometers per second, we can create a position in the **`TEME`** reference frame:

```
>>> from astropy.coordinates import TEME, CartesianDifferential,
CartesianRepresentation
>>> from astropy import units as u
>>> teme_p = CartesianRepresentation(teme_p*u.km)
>>> teme_v = CartesianDifferential(teme_v*u.km/u.s)
>>> teme = TEME(teme_p.with_differentials(teme_v), obstime=t)
```

Note how we are careful to set the observed time of the **TEME** frame to the time at which we calculated satellite position.

Transforming TEME to Other Coordinate Systems

Once you have satellite positions in **TEME** coordinates they can be transformed into any **astropy.coordinates** frame.

For example, to find the overhead latitude, longitude, and height of the satellite:

```
>>> from astropy.coordinates import ITRS
>>> itrs = teme.transform_to(ITRS(obstime=t))
>>> location = itrs.earth_location
>>> location.geodetic
GeodeticLocation(lon=<Longitude 160.34199789 deg>, lat=<Latitude
-24.6609379 deg>, height=<Quantity 420.17927591 km>)
```

Or, if you want to find the altitude and azimuth of the satellite from a particular location:

```
>>> from astropy.coordinates import EarthLocation, AltAz
>>> siding_spring = EarthLocation.of_site('aao')
>>> aa = teme.transform_to(AltAz(obstime=t, location=siding_spring))
>>> aa.alt
<Latitude 10.94798428 deg>
>>> aa.az
<Longitude 59.28807348 deg>
```

Formatting Coordinate Strings

Getting a string representation of a coordinate is most powerfully approached by treating the components (e.g., RA and Dec) separately.

Examples

To get the string representation of a coordinate:

```
>>> from astropy.coordinates import ICRS
>>> from astropy import units as u
>>> c = ICRS(187.70592*u.degree, 12.39112*u.degree)
>>> str(c.ra) + ' ' + str(c.dec)
'187d42m21.312s 12d23m28.032s'
```

To get better control over the formatting, you can use the angles' **`to_string()`** method (see Working with Angles for more). For example:

```
>>> rahmsstr = c.ra.to_string(u.hour)
>>> str(rahmsstr)
'12h30m49.4208s'
>>> decdmsstr = c.dec.to_string(u.degree, alwayssign=True)
>>> str(decdmsstr)
'+12d23m28.032s'
>>> rahmsstr + ' ' + decdmsstr
u'12h30m49.4208s +12d23m28.032s'
```

You can also use Python's **`format`** string method to create more complex string expressions, such as IAU-style coordinates or even full sentences:

```
>>> 'SDSS J{0}{1}'.format(c.ra.to_string(unit=u.hourangle, sep='',
precision=2, pad=True), c.dec.to_string(sep='', precision=2,
alwayssign=True, pad=True))
'SDSS J123049.42+122328.03'
>>> 'The galaxy M87, at an RA of {0.ra.hour:.2f} hours and Dec of
{0.dec.deg:.1f} degrees, has an impressive jet.'.format(c)
'The galaxy M87, at an RA of 12.51 hours and Dec of 12.4 degrees, has
an impressive jet.'
```

Separations, Offsets, Catalog Matching, and Related Functionality

`astropy.coordinates` contains commonly-used tools for comparing or matching coordinate objects. Of particular importance are those for determining separations between coordinates and those for matching a coordinate (or coordinates) to a catalog. These are mainly implemented as methods on the coordinate objects.

Separations

The on-sky separation can be computed with the **`astropy.coordinates.BaseCoordinateFrame.separation()`** or **`astropy.coordinates.SkyCoord.separation()`** methods, which computes the great-circle distance (*not* the small-angle approximation):

```
>>> import numpy as np
>>> from astropy import units as u
>>> from astropy.coordinates import SkyCoord
>>> c1 = SkyCoord('5h23m34.5s', '-69d45m22s', frame='icrs')
>>> c2 = SkyCoord('0h52m44.8s', '-72d49m43s', frame='fk5')
>>> sep = c1.separation(c2)
>>> sep
<Angle 20.74611448 deg>
```

The returned object is an **`Angle`** instance, so it is possible to access the angle in any of several equivalent angular units:

```
>>> sep.radian
0.36208800460262563
>>> sep.hour
1.3830742984029318
>>> sep.arcminute
1244.7668685626384
>>> sep.arcsecond
74686.0121137583
```

Also note that the two input coordinates were not in the same frame — one is automatically converted to match the other, ensuring that even though they are in different frames, the separation is determined consistently.

In addition to the on-sky separation described above, **`astropy.coordinates.BaseCoordinateFrame.separation_3d()`** or **`astropy.coordinates.SkyCoord.separation_3d()`** methods will determine the 3D distance between two coordinates that have `distance` defined:

```
>>> from astropy import units as u
>>> from astropy.coordinates import SkyCoord
>>> c1 = SkyCoord('5h23m34.5s', '-69d45m22s', distance=70*u.kpc,
frame='icrs')
>>> c2 = SkyCoord('0h52m44.8s', '-72d49m43s', distance=80*u.kpc,
frame='icrs')
>>> sep = c1.separation_3d(c2)
>>> sep
<Distance 28.74398816 kpc>
```

Offsets

Closely related to angular separations are offsets between coordinates. The key distinction for offsets is generally the concept of a “from” and “to” coordinate rather than the single scalar angular offset of a separation. `coordinates` contains conveniences to compute some of the common offsets encountered in astronomy.

The first piece of such functionality is the `position_angle()` method. This method computes the position angle between one `SkyCoord` instance and another (passed as the argument) following the astronomy convention (positive angles East of North):

```
>>> from astropy import units as u
>>> from astropy.coordinates import SkyCoord
>>> c1 = SkyCoord(1*u.deg, 1*u.deg, frame='icrs')
>>> c2 = SkyCoord(2*u.deg, 2*u.deg, frame='icrs')
>>> c1.position_angle(c2).to(u.deg)
<Angle 44.97818294 deg>
```

The combination of `separation()` and `position_angle()` thus give a set of directional offsets. To do the inverse operation — determining the new “destination” coordinate given a separation and position angle — the `directional_offset_by()` method is provided:

```
>>> from astropy import units as u
>>> from astropy.coordinates import SkyCoord
>>> c1 = SkyCoord(1*u.deg, 1*u.deg, frame='icrs')
>>> position_angle = 45 * u.deg
>>> separation = 1.414 * u.deg
>>> c1.directional_offset_by(position_angle, separation)
<SkyCoord (ICRS): (ra, dec) in deg
    (2.0004075, 1.99964588)>
```

This technique is also useful for computing the midpoint (or indeed any point) between two coordinates in a way that accounts for spherical geometry (i.e., instead of averaging the RAs/Decs separately):

```
>>> from astropy import units as u
>>> from astropy.coordinates import SkyCoord
>>> coord1 = SkyCoord(0*u.deg, 0*u.deg, frame='icrs')
>>> coord2 = SkyCoord(1*u.deg, 1*u.deg, frame='icrs')
>>> pa = coord1.position_angle(coord2)
>>> sep = coord1.separation(coord2)
>>> coord1.directional_offset_by(pa, sep/2)
<SkyCoord (ICRS): (ra, dec) in deg
    (0.49996192, 0.50001904)>
```

There is also a **spherical_offsets_to()** method for computing angular offsets (e.g., small shifts like you might give a telescope operator to move from a bright star to a fainter target):

```
>>> from astropy import units as u
>>> from astropy.coordinates import SkyCoord
>>> bright_star = SkyCoord('8h50m59.75s', '+11d39m22.15s',
frame='icrs')
>>> faint_galaxy = SkyCoord('8h50m47.92s', '+11d39m32.74s',
frame='icrs')
>>> dra, ddec = bright_star.spherical_offsets_to(faint_galaxy)
>>> dra.to(u.arcsec)
<Angle -173.78873354 arcsec>
>>> ddec.to(u.arcsec)
<Angle 10.60510342 arcsec>
```

“Sky Offset” Frames

To extend the concept of spherical offsets, **coordinates** has a frame class **SkyOffsetFrame** which creates distinct frames that are centered on a specific point. These are known as “sky offset frames,” as they are a convenient way to create a frame centered on an arbitrary position on the sky suitable for computing positional offsets (e.g., for astrometry):

```
>>> from astropy import units as u
>>> from astropy.coordinates import SkyOffsetFrame, ICRS, SkyCoord
>>> center = ICRS(10*u.deg, 45*u.deg)
>>> center.transform_to(SkyOffsetFrame(origin=center))
<SkyOffsetICRS Coordinate (rotation=0.0 deg, origin=<ICRS Coordinate:
(ra, dec) in deg
    (10., 45.)>): (lon, lat) in deg
    (0., 0.)>
>>> target = ICRS(11*u.deg, 46*u.deg)
>>> target.transform_to(SkyOffsetFrame(origin=center))
<SkyOffsetICRS Coordinate (rotation=0.0 deg, origin=<ICRS Coordinate:
(ra, dec) in deg
    (10., 45.)>): (lon, lat) in deg
    (0.69474685, 1.00428706)>
```

Alternatively, the convenience method **skyoffset_frame()** lets you create a sky offset frame from an existing **SkyCoord**:

```
>>> center = SkyCoord(10*u.deg, 45*u.deg)
>>> aframe = center.skyoffset_frame()
>>> target.transform_to(aframe)
<SkyOffsetICRS Coordinate (rotation=0.0 deg, origin=<ICRS Coordinate:
(ra, dec) in deg
```

```
    (10., 45.)>): (lon, lat) in deg
    (0.69474685, 1.00428706)>
>>> other = SkyCoord(9*u.deg, 44*u.deg, frame='fk5')
>>> other.transform_to(aframe)
<SkyCoord (SkyOffsetICRS: rotation=0.0 deg, origin=<ICRS Coordinate:
(ra, dec) in deg
    (10., 45.)>): (lon, lat) in deg
    (-0.71943945, -0.99556216)>
```

Note

While sky offset frames *appear* to be all the same class, this not the case: the sky offset frame for each different type of frame for `origin` is actually a distinct class. E.g., `SkyOffsetFrame(origin=ICRS(...))` yields an object of class `SkyOffsetICRS`, *not* `SkyOffsetFrame`. While this is not important for most uses of this class, it is important for things like type-checking, because something like `SkyOffsetFrame(origin=ICRS(...)).__class__ is SkyOffsetFrame` will *not* be `True`, as it would be for most classes.

This same frame is also useful as a tool for defining frames that are relative to a specific, known object useful for hierarchical physical systems like galaxy groups. For example, objects around M31 are sometimes shown in a coordinate frame aligned with standard ICRA RA/Dec, but on M31:

```
>>> m31 = SkyCoord(10.6847083*u.deg, 41.26875*u.deg, frame='icrs')
>>> ngc147 = SkyCoord(8.3005*u.deg, 48.5087389*u.deg, frame='icrs')
>>> ngc147_inm31 = ngc147.transform_to(m31.skyoffset_frame())
>>> xi, eta = ngc147_inm31.lon, ngc147_inm31.lat
>>> xi
<Longitude -1.59206948 deg>
>>> eta
<Latitude 7.26183757 deg>
```

Note

Currently, distance information in the `origin` of a **`SkyOffsetFrame`** is not used to compute any part of the transform. The `origin` is only used for on-sky rotation. This may change in the future, however.

Matching Catalogs

`coordinates` leverages the coordinate framework to make it possible to find the closest coordinates in a catalog to a desired set of other coordinates. For example, assuming `ra1` / `dec1` and `ra2` / `dec2` are NumPy arrays loaded from some file:

```
>>> from astropy.coordinates import SkyCoord
>>> from astropy import units as u
>>> c = SkyCoord(ra=ra1*u.degree, dec=dec1*u.degree)
>>> catalog = SkyCoord(ra=ra2*u.degree, dec=dec2*u.degree)
>>> idx, d2d, d3d = c.match_to_catalog_sky(catalog)
```

The distances returned `d3d` are 3-dimensional distances. Unless both source (`c`) and catalog (`catalog`) coordinates have associated distances, this quantity assumes that all sources are at a distance of 1 (dimensionless).

You can also find the nearest 3D matches, different from the on-sky separation shown above only when the coordinates were initialized with a `distance` :

```
>>> c = SkyCoord(ra=ra1*u.degree, dec=dec1*u.degree,
distance=distance1*u.kpc)
>>> catalog = SkyCoord(ra=ra2*u.degree, dec=dec2*u.degree,
distance=distance2*u.kpc)
>>> idx, d2d, d3d = c.match_to_catalog_3d(catalog)
```

Now `idx` are indices into `catalog` that are the closest objects to each of the coordinates in `c` , `d2d` are the on-sky distances between them, and `d3d` are the 3-dimensional distances. Because coordinate objects support indexing, `idx` enables easy access to the matched set of coordinates in the catalog:

```
>>> matches = catalog[idx]
>>> (matches.separation_3d(c) == d3d).all()
True
>>> dra, ddec = c.spherical_offsets_to(matches)
```

This functionality can also be accessed from the **`match_coordinates_sky()`** and **`match_coordinates_3d()`** functions. These will work on either **`SkyCoord`** objects *or* the lower-level frame classes:

```
>>> from astropy.coordinates import match_coordinates_sky
>>> idx, d2d, d3d = match_coordinates_sky(c, catalog)
>>> idx, d2d, d3d = match_coordinates_sky(c.frame, catalog.frame)
```

It is possible to impose a separation constraint (e.g., the maximum separation to be considered a match) by creating a boolean mask with `d2d` or `d3d` . For example:

```
>>> max_sep = 1.0 * u.arcsec
>>> idx, d2d, d3d = c.match_to_catalog_3d(catalog)
>>> sep_constraint = d2d < max_sep
>>> c_matches = c[sep_constraint]
>>> catalog_matches = catalog[idx[sep_constraint]]
```

Now, `c_matches` and `catalog_matches` are the matched sources in `c` and `catalog`, respectively, which are separated by less than 1 arcsecond.

Searching around Coordinates

Closely related functionality can be used to search for *all* coordinates within a certain distance (either 3D distance or on-sky) of another set of coordinates. The `search_around_*` methods (and functions) provide this functionality, with an interface very similar to `match_coordinates_*`:

```
>>> import numpy as np
>>> idxc, idxcatalog, d2d, d3d = catalog.search_around_sky(c,
1*u.deg)
>>> np.all(d2d < 1*u.deg)
True
```

```
>>> idxc, idxcatalog, d2d, d3d = catalog.search_around_3d(c, 1*u.kpc)
>>> np.all(d3d < 1*u.kpc)
True
```

The key difference for these methods is that there can be multiple (or no) matches in `catalog` around any locations in `c`. Hence, indices into both `c` and `catalog` are returned instead of just indices into `catalog`. These can then be indexed back into the two **SkyCoord** objects, or, for that matter, any array with the same order:

```
>>> np.all(c[idxc].separation(catalog[idxcatalog]) == d2d)
True
>>> np.all(c[idxc].separation_3d(catalog[idxcatalog]) == d3d)
True
>>> print(catalog_objectnames[idxcatalog])
['NGC 1234' 'NGC 4567' ...]
```

Note, though, that this dual-indexing means that `search_around_*` does not work well if one of the coordinates is a scalar, because the returned index would not make sense for a scalar:

```
>>> scalarc = SkyCoord(ra=1*u.deg, dec=2*u.deg,
distance=distance1*u.kpc)
>>> idxscalarc, idxcatalog, d2d, d3d =
catalog.search_around_sky(scalarc, 1*u.deg)
ValueError: One of the inputs to search_around_sky is a scalar.
```

As a result (and because the `search_around_*` algorithm is inefficient in the scalar case), the best approach for this scenario is to instead use the `separation*` methods:

```
>>> d2d = scalarc.separation(catalog)
>>> catalogmsk = d2d < 1*u.deg
>>> d3d = scalarc.separation_3d(catalog)
>>> catalog3dmsk = d3d < 1*u.kpc
```

The resulting `catalogmsk` or `catalog3dmsk` variables are boolean arrays rather than arrays of indices, but in practice they usually can be used in the same way as `idxcatalog` from the above examples. If you definitely do need indices instead of boolean masks, you can do:

```
>>> idxcatalog = np.where(catalogmsk)[0]
>>> idxcatalog3d = np.where(catalog3dmsk)[0]
```

Using and Designing Coordinate Representations

Points in a 3D vector space can be represented in different ways, such as Cartesian, spherical polar, cylindrical, and so on. These underlie the way coordinate data in **`astropy.coordinates`** is represented, as described in the Overview of astropy.coordinates Concepts. Below, we describe how you can use them on their own as a way to convert between different representations, including ones not built-in, and to do simple vector arithmetic.

The built-in representation classes are:

- **`CartesianRepresentation`**: Cartesian coordinates `x`, `y`, and `z`.
- **`SphericalRepresentation`**: spherical polar coordinates represented by a longitude (`lon`), a latitude (`lat`), and a distance (`distance`). The latitude is a value ranging from -90 to 90 degrees.
- **`UnitSphericalRepresentation`**: spherical polar coordinates on a unit sphere, represented by a longitude (`lon`) and latitude (`lat`).
- **`PhysicsSphericalRepresentation`**: spherical polar coordinates, represented by an inclination (`theta`) and azimuthal angle (`phi`), and radius `r`. The inclination goes from 0 to 180 degrees, and is related to the

latitude in the **SphericalRepresentation** by `theta = 90 deg - lat`.

- **CylindricalRepresentation**: cylindrical polar coordinates, represented by a cylindrical radius (`rho`), azimuthal angle (`phi`), and height (`z`).

Note

For information about using and changing the representation of **SkyCoord** objects, see the Representations section.

Instantiating and Converting

Representation classes are instantiated with **Quantity** objects:

```
>>> from astropy import units as u
>>> from astropy.coordinates.representation import
CartesianRepresentation
>>> car = CartesianRepresentation(3 * u.kpc, 5 * u.kpc, 4 * u.kpc)
>>> car
<CartesianRepresentation (x, y, z) in kpc
    (3., 5., 4.)>
```

Array **Quantity** objects can also be passed to representations. They will have the expected shape, which can be changed using methods with the same names as those for **ndarray**, such as `reshape`, `ravel`, etc.:

```
>>> x = u.Quantity([[1., 0., 0.], [3., 5., 3.]], u.m)
>>> y = u.Quantity([[0., 2., 0.], [4., 0., -4.]], u.m)
>>> z = u.Quantity([[0., 0., 3.], [0., 12., -12.]], u.m)
>>> car_array = CartesianRepresentation(x, y, z)
>>> car_array
<CartesianRepresentation (x, y, z) in m
    [[(1.,  0.,   0.), (0.,  2.,   0.), (0.,  0.,   3.)],
     [(3.,  4.,   0.), (5.,  0.,  12.), (3., -4., -12.)]]>
>>> car_array.shape
(2, 3)
>>> car_array.ravel()
<CartesianRepresentation (x, y, z) in m
    [(1.,  0.,   0.), (0.,  2.,   0.), (0.,  0.,   3.), (3.,  4.,
0.),
     (5.,  0.,  12.), (3., -4., -12.)]>
```

Representations can be converted to other representations using the `represent_as` method:

```
>>> from astropy.coordinates.representation import
SphericalRepresentation, CylindricalRepresentation
>>> sph = car.represent_as(SphericalRepresentation)
>>> sph
<SphericalRepresentation (lon, lat, distance) in (rad, rad, kpc)
    (1.03037683, 0.60126422, 7.07106781)>
>>> cyl = car.represent_as(CylindricalRepresentation)
>>> cyl
<CylindricalRepresentation (rho, phi, z) in (kpc, rad, kpc)
    (5.83095189, 1.03037683, 4.)>
```

All representations can be converted to each other without loss of information, with the exception of **`UnitSphericalRepresentation`**. This class is used to store the longitude and latitude of points but does not contain any distance to the points, and assumes that they are located on a unit and dimensionless sphere:

```
>>> from astropy.coordinates.representation import
UnitSphericalRepresentation
>>> sph_unit = car.represent_as(UnitSphericalRepresentation)
>>> sph_unit
<UnitSphericalRepresentation (lon, lat) in rad
    (1.03037683, 0.60126422)>
```

Converting back to Cartesian, the absolute scaling information has been removed, and the points are still located on a unit sphere:

```
>>> sph_unit = car.represent_as(UnitSphericalRepresentation)
>>> sph_unit.represent_as(CartesianRepresentation)
<CartesianRepresentation (x, y, z) [dimensionless]
    (0.42426407, 0.70710678, 0.56568542)>
```

Array Values and NumPy Array Method Analogs

Array **`Quantity`** objects can also be passed to representations, and such representations can be sliced, reshaped, etc., using the same methods as are available to **`ndarray`**. Similarly, on `numpy` version 1.17 and later, corresponding functions as well as others that affect the shape, such as **`atleast_1d`** and **`rollaxis`**, work as expected.

Example

To pass array **Quantity** objects to representations:

```
>>> import numpy as np
>>> x = np.linspace(0., 5., 6)
>>> y = np.linspace(10., 15., 6)
>>> z = np.linspace(20., 25., 6)
>>> car_array = CartesianRepresentation(x * u.m, y * u.m, z * u.m)
>>> car_array
<CartesianRepresentation (x, y, z) in m
    [(0., 10., 20.), (1., 11., 21.), (2., 12., 22.),
     (3., 13., 23.), (4., 14., 24.), (5., 15., 25.)]>
```

To manipulate using methods and `numpy` functions:

```
.. doctest-requires:: numpy>=1.17
```

```
>>> car_array.reshape(3, 2)
<CartesianRepresentation (x, y, z) in m
    [[(0., 10., 20.), (1., 11., 21.)],
     [(2., 12., 22.), (3., 13., 23.)],
     [(4., 14., 24.), (5., 15., 25.)]]>
>>> car_array[2]
<CartesianRepresentation (x, y, z) in m
    (2., 12., 22.)>
>>> car_array[2] = car_array[1]
>>> car_array[:3]
<CartesianRepresentation (x, y, z) in m
    [(0., 10., 20.), (1., 11., 21.), (1., 11., 21.)]>
>>> np.roll(car_array, 1)
<CartesianRepresentation (x, y, z) in m
    [(5., 15., 25.), (0., 10., 20.), (1., 11., 21.), (1., 11., 21.),
     (3., 13., 23.), (4., 14., 24.)]>
```

And to set elements using other representation classes (as long as they are compatible in their units and number of dimensions):

```
>>> car_array[2] = SphericalRepresentation(0*u.deg, 0*u.deg, 99*u.m)
>>> car_array[:3]
<CartesianRepresentation (x, y, z) in m
    [(0., 10., 20.), (1., 11., 21.), (99., 0., 0.)]>
>>> car_array[0] = UnitSphericalRepresentation(0*u.deg, 0*u.deg)
Traceback (most recent call last):
...
ValueError: value must be representable as CartesianRepresentation
without loss of information.
```

Vector Arithmetic

Representations support basic vector arithmetic such as taking the norm, multiplying with and dividing by quantities, and taking dot and cross products, as well as adding, subtracting, summing and taking averages of representations, and multiplying with matrices.

Note

All arithmetic except the matrix multiplication works with non-Cartesian representations as well. For taking the norm, multiplication, and division, this uses just the non-angular components, while for the other operations the representation is converted to Cartesian internally before the operation is done, and the result is converted back to the original representation. Hence, for optimal speed it may be best to work using Cartesian representations.

Examples

To see how vector arithmetic operations work with representation objects, consider the following examples:

```
>>> car_array = CartesianRepresentation([[1., 0., 0.], [3., 5.,  3.]]
* u.m,
...                                        [[0., 2., 0.], [4., 0., -4.]]
* u.m,
...                                        [[0., 0., 3.], [0.,12.,-12.]]
* u.m)
>>> car_array
<CartesianRepresentation (x, y, z) in m
    [[(1.,  0.,  0.), (0.,  2.,   0.), (0.,  0.,   3.)],
     [(3.,  4.,  0.), (5.,  0.,  12.), (3., -4., -12.)]]>
>>> car_array.norm()
<Quantity [[ 1.,  2.,  3.],
           [ 5., 13., 13.]] m>
>>> car_array / car_array.norm()
<CartesianRepresentation (x, y, z) [dimensionless]
    [[(1.        ,  0.        ,  0.        ),
      (0.        ,  1.        ,  0.        ),
      (0.        ,  0.        ,  1.        )],
     [(0.6       ,  0.8       ,  0.        ),
      (0.38461538,  0.        ,  0.92307692),
      (0.23076923, -0.30769231, -0.92307692)]]>
>>> (car_array[1] - car_array[0]) / (10. * u.s)
<CartesianRepresentation (x, y, z) in m / s
    [(0.2,  0.4,  0. ), (0.5, -0.2,  1.2), (0.3, -0.4, -1.5)]>
>>> car_array.sum()
<CartesianRepresentation (x, y, z) in m
    (12.,  2.,  3.)>
```

```
>>> car_array.mean(axis=0)
<CartesianRepresentation (x, y, z) in m
    [(2. ,  2.,  0. ), (2.5,  1.,  6. ), (1.5, -2., -4.5)]>

>>> unit_x = UnitSphericalRepresentation(0.*u.deg, 0.*u.deg)
>>> unit_y = UnitSphericalRepresentation(90.*u.deg, 0.*u.deg)
>>> unit_z = UnitSphericalRepresentation(0.*u.deg, 90.*u.deg)
>>> car_array.dot(unit_x)
<Quantity [[1., 0., 0.],
           [3., 5., 3.]] m>
>>> car_array.dot(unit_y)
<Quantity [[ 6.12323400e-17,  2.00000000e+00,  0.00000000e+00],
           [ 4.00000000e+00,  3.06161700e-16, -4.00000000e+00]] m>
>>> car_array.dot(unit_z)
<Quantity [[ 6.12323400e-17,  0.00000000e+00,  3.00000000e+00],
           [ 1.83697020e-16,  1.20000000e+01, -1.20000000e+01]] m>
>>> car_array.cross(unit_x)
<CartesianRepresentation (x, y, z) in m
    [[(0.,  0.,  0.), (0.,   0., -2.), (0.,   3.,  0.)],
     [(0.,  0., -4.), (0.,  12.,  0.), (0., -12.,  4.)]]>

>>> from astropy.coordinates.matrix_utilities import rotation_matrix
>>> rotation = rotation_matrix(90 * u.deg, axis='z')
>>> rotation
array([[ 6.12323400e-17,  1.00000000e+00,  0.00000000e+00],
       [-1.00000000e+00,  6.12323400e-17,  0.00000000e+00],
       [ 0.00000000e+00,  0.00000000e+00,  1.00000000e+00]])
>>> car_array.transform(rotation)
<CartesianRepresentation (x, y, z) in m
    [[( 6.12323400e-17, -1.00000000e+00,   0.),
      ( 2.00000000e+00,  1.22464680e-16,   0.),
      ( 0.00000000e+00,  0.00000000e+00,   3.)],
     [( 4.00000000e+00, -3.00000000e+00,   0.),
      ( 3.06161700e-16, -5.00000000e+00,  12.),
      (-4.00000000e+00, -3.00000000e+00, -12.)]]>
```

Differentials and Derivatives of Representations

In addition to positions in 3D space, coordinates also deal with proper motions and radial velocities, which require a way to represent differentials of coordinates (i.e., finite realizations) of derivatives. To support this, the representations all have corresponding `Differential` classes, which can hold offsets or derivatives in terms of the components of the representation class. Adding such an offset to a representation means the offset is taken in the direction of the corresponding coordinate. (Although for any representation other than Cartesian, this is only defined relative to a specific location, as the unit vectors are not invariant.)

Examples

To see how the `Differential` classes of representations works, consider the following:

```
>>> from astropy.coordinates import SphericalRepresentation,
SphericalDifferential
>>> sph_coo = SphericalRepresentation(lon=0.*u.deg, lat=0.*u.deg,
...                                       distance=1.*u.kpc)
>>> sph_derivative = SphericalDifferential(d_lon=1.*u.arcsec/u.yr,
...                                            d_lat=0.*u.arcsec/u.yr,
...                                            d_distance=0.*u.km/u.s)
>>> sph_derivative.to_cartesian(base=sph_coo)
<CartesianRepresentation (x, y, z) in arcsec kpc / (rad yr)
    (0., 1., 0.)>
```

Note how the conversion to Cartesian can only be done using a `base`, since otherwise the code cannot know what direction an increase in longitude corresponds to. For `lon=0`, this is in the `y` direction. Now, to get the coordinates at two later times:

```
>>> sph_coo + sph_derivative * [1., 3600*180/np.pi] * u.yr
<SphericalRepresentation (lon, lat, distance) in (rad, rad, kpc)
    [(4.84813681e-06, 0., 1.        ), (7.85398163e-01, 0.,
1.41421356)]>
```

The above shows how addition is not to longitude itself, but in the direction of increasing longitude: for the large shift, by the equivalent of one radian, the distance has increased as well (after all, a source will likely not move along a curve on the sky!). This also means that the order of operations is important:

```
>>> big_offset = SphericalDifferential(1.*u.radian, 0.*u.radian,
0.*u.kpc)
>>> sph_coo + big_offset + big_offset
<SphericalRepresentation (lon, lat, distance) in (rad, rad, kpc)
    (1.57079633, 0., 2.)>
>>> sph_coo + (big_offset + big_offset)
<SphericalRepresentation (lon, lat, distance) in (rad, rad, kpc)
    (1.10714872, 0., 2.23606798)>
```

Often, you may have just a proper motion or a radial velocity, but not both:

```
>>> from astropy.coordinates import UnitSphericalDifferential,
RadialDifferential
>>> radvel = RadialDifferential(1000*u.km/u.s)
>>> sph_coo + radvel * 1. * u.Myr
<SphericalRepresentation (lon, lat, distance) in (rad, rad, kpc)
    (0., 0., 2.02271217)>
>>> pm = UnitSphericalDifferential(1.*u.mas/u.yr, 0.*u.mas/u.yr)
>>> sph_coo + pm * 1. * u.Myr
<SphericalRepresentation (lon, lat, distance) in (rad, rad, kpc)
    (0.0048481, 0., 1.00001175)>
>>> pm + radvel
<SphericalDifferential (d_lon, d_lat, d_distance) in (mas / yr, mas /
yr, km / s)
    (1., 0., 1000.)>
>>> sph_coo + (pm + radvel) * 1. * u.Myr
<SphericalRepresentation (lon, lat, distance) in (rad, rad, kpc)
    (0.00239684, 0., 2.02271798)>
```

Note in the above that the proper motion is defined strictly as a change in longitude (i.e., it does not include a `cos(latitude)` term). There are special classes where this term is included:

```
>>> from astropy.coordinates import UnitSphericalCosLatDifferential
>>> sph_lat60 = SphericalRepresentation(lon=0.*u.deg, lat=60.*u.deg,
...                                         distance=1.*u.kpc)
>>> pm = UnitSphericalDifferential(1.*u.mas/u.yr, 0.*u.mas/u.yr)
>>> pm
<UnitSphericalDifferential (d_lon, d_lat) in mas / yr
    (1., 0.)>
>>> pm_coslat = UnitSphericalCosLatDifferential(1.*u.mas/u.yr,
0.*u.mas/u.yr)
>>> pm_coslat
<UnitSphericalCosLatDifferential (d_lon_coslat, d_lat) in mas / yr
    (1., 0.)>
>>> sph_lat60 + pm * 1. * u.Myr
<SphericalRepresentation (lon, lat, distance) in (rad, rad, kpc)
    (0.0048481, 1.04719246, 1.00000294)>
>>> sph_lat60 + pm_coslat * 1. * u.Myr
<SphericalRepresentation (lon, lat, distance) in (rad, rad, kpc)
    (0.00969597, 1.0471772, 1.00001175)>
```

Close inspections shows that indeed the changes are as expected. The systems with and without `cos(latitude)` can be converted to each other, provided you supply the `base` (representation):

```
>>> usph_lat60 = sph_lat60.represent_as(UnitSphericalRepresentation)
>>> pm_coslat2 = pm.represent_as(UnitSphericalCosLatDifferential,
...                                  base=usph_lat60)
```

```
>>> pm_coslat2
<UnitSphericalCosLatDifferential (d_lon_coslat, d_lat) in mas / yr
    (0.5, 0.)>
>>> sph_lat60 + pm_coslat2 * 1. * u.Myr
<SphericalRepresentation (lon, lat, distance) in (rad, rad, kpc)
    (0.0048481, 1.04719246, 1.00000294)>
```

Note

At present, the differential classes are generally meant to work with first derivatives, but they do not check the units of the inputs to enforce this. Passing in second derivatives (e.g., acceleration values with acceleration units) will succeed, but any transformations that occur through re-representation of the differential will not necessarily be correct.

Attaching `Differential` Objects to `Representation` Objects

Warning

The API for this functionality may change in future versions and should be viewed as provisional!

`Differential` objects can be attached to `Representation` objects as a way to encapsulate related information into a single object. `Differential` objects can be passed in to the initializer of any of the built-in `Representation` classes.

Example

To store a single velocity differential with a position:

```
>>> from astropy.coordinates import representation as r
>>> dif = r.SphericalDifferential(d_lon=1 * u.mas/u.yr,
...                               d_lat=2 * u.mas/u.yr,
...                               d_distance=3 * u.km/u.s)
>>> rep = r.SphericalRepresentation(lon=0.*u.deg, lat=0.*u.deg,
...                                 distance=1.*u.kpc,
...                                 differentials=dif)
>>> rep
<SphericalRepresentation (lon, lat, distance) in (deg, deg, kpc)
    (0., 0., 1.)
 (has differentials w.r.t.: 's')>
>>> rep.differentials
{'s': <SphericalDifferential (d_lon, d_lat, d_distance) in (mas / yr,
mas / yr, km / s)
     (1., 2., 3.)>}
```

The `Differential` objects are stored as a Python dictionary on the `Representation` object with keys equal to the (string) unit with which the differential derivatives are taken (converted to SI).

In this case the key is `'s'` (second) because the `Differential` units are velocities, a time derivative. Passing a single differential to the `Representation` initializer will automatically generate the necessary key and store it in the differentials dictionary, but a dictionary is required to specify multiple differentials:

```
>>> dif2 = r.SphericalDifferential(d_lon=4 * u.mas/u.yr**2,
...                                d_lat=5 * u.mas/u.yr**2,
...                                d_distance=6 * u.km/u.s**2)
>>> rep = r.SphericalRepresentation(lon=0.*u.deg, lat=0.*u.deg,
...                                 distance=1.*u.kpc,
...                                 differentials={'s': dif, 's2':
dif2})
>>> rep.differentials['s']
<SphericalDifferential (d_lon, d_lat, d_distance) in (mas / yr, mas /
yr, km / s)
    (1., 2., 3.)>
>>> rep.differentials['s2']
<SphericalDifferential (d_lon, d_lat, d_distance) in (mas / yr2, mas
/ yr2, km / s2)
    (4., 5., 6.)>
```

`Differential` objects can also be attached to a `Representation` after creation:

```
>>> rep = r.CartesianRepresentation(x=1 * u.kpc, y=2 * u.kpc, z=3 *
u.kpc)
>>> dif = r.CartesianDifferential(*[1, 2, 3] * u.km/u.s)
>>> rep = rep.with_differentials(dif)
>>> rep
<CartesianRepresentation (x, y, z) in kpc
    (1., 2., 3.)
 (has differentials w.r.t.: 's')>
```

This works for array data as well, as long as the shape of the `Differential` data is the same as that of the `Representation`:

```
>>> xyz = np.arange(12).reshape(3, 4) * u.au
>>> d_xyz = np.arange(12).reshape(3, 4) * u.km/u.s
>>> rep = r.CartesianRepresentation(*xyz)
>>> dif = r.CartesianDifferential(*d_xyz)
>>> rep = rep.with_differentials(dif)
>>> rep
<CartesianRepresentation (x, y, z) in AU
    [(0., 4.,  8.), (1., 5.,  9.), (2., 6., 10.), (3., 7., 11.)]
 (has differentials w.r.t.: 's')>
```

As with a `Representation` instance without a differential, to convert the positional data to a new representation, use the `.represent_as()`:

```
>>> rep.represent_as(r.SphericalRepresentation)
<SphericalRepresentation (lon, lat, distance) in (rad, rad, AU)
    [(1.57079633, 1.10714872,  8.94427191),
     (1.37340077, 1.05532979, 10.34408043),
     (1.24904577, 1.00685369, 11.83215957),
     (1.16590454, 0.96522779, 13.37908816)]>
```

However, by passing just the desired representation class, only the `Representation` has changed, and the differentials are dropped. To re-represent both the `Representation` and any `Differential` objects, you must specify target classes for the `Differential` as well:

```
>>> rep2 = rep.represent_as(r.SphericalRepresentation,
r.SphericalDifferential)
>>> rep2
<SphericalRepresentation (lon, lat, distance) in (rad, rad, AU)
  [(1.57079633, 1.10714872,  8.94427191),
   (1.37340077, 1.05532979, 10.34408043),
   (1.24904577, 1.00685369, 11.83215957),
   (1.16590454, 0.96522779, 13.37908816)]
 (has differentials w.r.t.: 's')>
>>> rep2.differentials['s']
```

```
<SphericalDifferential (d_lon, d_lat, d_distance) in (km rad / (AU
s), km rad / (AU s), km / s)
    [( 6.12323400e-17, 1.11022302e-16,  8.94427191),
     (-2.77555756e-17, 5.55111512e-17, 10.34408043),
     ( 0.00000000e+00, 0.00000000e+00, 11.83215957),
     ( 5.55111512e-17, 0.00000000e+00, 13.37908816)]>
```

Shape-changing operations (e.g., reshapes) are propagated to all `Differential` objects because they are guaranteed to have the same shape as their host `Representation` object:

```
>>> rep.shape
(4,)
>>> rep.differentials['s'].shape
(4,)
>>> new_rep = rep.reshape(2, 2)
>>> new_rep.shape
(2, 2)
>>> new_rep.differentials['s'].shape
(2, 2)
```

This also works for slicing:

```
>>> new_rep = rep[:2]
>>> new_rep.shape
(2,)
>>> new_rep.differentials['s'].shape
(2,)
```

Operations on representations that return **`Quantity`** objects (as opposed to other `Representation` instances) still work, but only operate on the positional information, for example:

```
>>> rep.norm()
<Quantity [ 8.94427191, 10.34408043, 11.83215957, 13.37908816] AU>
```

Operations that involve combining or scaling representations or pairs of representation objects that contain differentials will currently fail, but support for some operations may be added in future versions:

```
>>> rep + rep
Traceback (most recent call last):
...
TypeError: Operation 'add' is not supported when differentials are
attached to a CartesianRepresentation.
```

If you have a `Representation` with attached `Differential` objects, you can retrieve a copy of the `Representation` without the `Differential` object and use this `Differential` -free object for any arithmetic operation:

```
>>> 15 * rep.without_differentials()
<CartesianRepresentation (x, y, z) in AU
    [( 0.,  60., 120.), (15.,  75., 135.), (30.,  90., 150.),
     (45., 105., 165.)]>
```

Creating Your Own Representations

To create your own representation class, your class must inherit from the **`BaseRepresentation`** class. This base has an `__init__` method that will put all arguments components through their initializers, verify they can be broadcast against each other, and store the components on `self` as the name prefixed with '_'. Furthermore, through its metaclass it provides default properties for the components so that they can be accessed using `<instance>.<component>`. For the machinery to work, the following must be defined:

- `attr_classes` class attribute (`OrderedDict`):

  Defines through its keys the names of the components (as well as the default order), and through its values defines the class of which they should be instances (which should be **`Quantity`** or a subclass, or anything that can initialize it).

- `from_cartesian` class method:

  Takes a **`CartesianRepresentation`** object and returns an instance of your class.

- `to_cartesian` method:

  Returns a **`CartesianRepresentation`** object.

- `__init__` method (optional):

  If you want more than the basic initialization and checks provided by the base representation class, or just an explicit signature, you can define your own `__init__`. In general, it is recommended to stay close to the signature assumed by the base representation, `__init__(self, comp1, comp2, comp3, copy=True)`, and use `super` to call the base representation initializer.

Once you do this, you will then automatically be able to call `represent_as` to convert other representations to/from your representation class. Your representation will also be available for use in **SkyCoord** and all frame classes.

A representation class may also have a `_unit_representation` attribute (although it is not required). This attribute points to the appropriate “unit” representation (i.e., a representation that is dimensionless). This is probably only meaningful for subclasses of **SphericalRepresentation**, where it is assumed that it will be a subclass of **UnitSphericalRepresentation**.

Finally, if you wish to also use offsets in your coordinate system, two further methods should be defined (please see **SphericalRepresentation** for an example):

- `unit_vectors` method:

  Returns a `dict` with a **CartesianRepresentation** of unit vectors in the direction of each component.

- `scale_factors` method:

  Returns a `dict` with a **Quantity** for each component with the appropriate physical scale factor for a unit change in that direction.

And furthermore you should define a `Differential` class based on **BaseDifferential**. This class only needs to define:

- `base_representation` attribute:

  A link back to the representation for which this differential holds.

In pseudo-code, this means that a class will look like:

```
class MyRepresentation(BaseRepresentation):

    attr_classes = OrderedDict([('comp1', ComponentClass1),
                                ('comp2', ComponentClass2),
                                ('comp3', ComponentClass3)])

    # __init__ is optional
    def __init__(self, comp1, comp2, comp3, copy=True):
        super().__init__(comp1, comp2, comp3, copy=copy)
        ...

    @classmethod
    def from_cartesian(self, cartesian):
        ...
        return MyRepresentation(...)

    def to_cartesian(self):
        ...
        return CartesianRepresentation(...)

    # if differential motion is needed
    def unit_vectors(self):
        ...
        return {'comp1': CartesianRepresentation(...),
                'comp2': CartesianRepresentation(...),
                'comp3': CartesianRepresentation(...)}

    def scale_factors(self):
        ...
        return {'comp1': ...,
                'comp2': ...,
                'comp3': ...}

class MyDifferential(BaseDifferential):
    base_representation = MyRepresentation
```

Using and Designing Coordinate Frames

In **`astropy.coordinates`**, as outlined in the Overview of astropy.coordinates Concepts, subclasses of **`BaseCoordinateFrame`** (“frame classes”) define particular coordinate frames. They can (but do not *have* to) contain representation objects storing the actual coordinate data. The actual coordinate transformations are defined as functions that transform representations between frame classes. This approach serves to separate high-level user functionality (see Using the SkyCoord High-Level Class) and details of how the coordinates are actually stored (see Using and Designing Coordinate Representations) from the definition of frames and how they are transformed.

Using Frame Objects

Frames without Data

Frame objects have two distinct (but related) uses. The first is storing the information needed to uniquely define a frame (e.g., equinox, observation time). This information is stored on the frame objects as (read-only) Python attributes, which are set when the object is first created:

```
>>> from astropy.coordinates import ICRS, FK5
>>> FK5(equinox='J1975')
<FK5 Frame (equinox=J1975.000)>
>>> ICRS()  # has no attributes
<ICRS Frame>
```

```
>>> FK5()  # uses default equinox
<FK5 Frame (equinox=J2000.000)>
```

The specific names of attributes available for a particular frame (and their default values) are available as the class method `get_frame_attr_names`:

```
>>> FK5.get_frame_attr_names()
OrderedDict([('equinox', <Time object: scale='tt' format='jyear_str'
value=J2000.000>)])
```

You can access any of the attributes on a frame by using standard Python attribute access. Note that for cases like `equinox`, which are time inputs, if you pass in any unambiguous time string, it will be converted into an **`Time`** object (see Inferring Input Format):

```
>>> f = FK5(equinox='J1975')
>>> f.equinox
<Time object: scale='tt' format='jyear_str' value=J1975.000>
>>> f = FK5(equinox='2011-05-15T12:13:14')
>>> f.equinox
<Time object: scale='utc' format='isot'
value=2011-05-15T12:13:14.000>
```

Frames with Data

The second use for frame objects is to store actual realized coordinate data for frames like those described above. In this use, it is similar to the **`SkyCoord`** class, and in fact, the **`SkyCoord`** class internally uses the frame classes as its implementation. However, the frame classes have fewer “convenience” features, thereby streamlining the implementation of frame classes. As such, they are created similarly to **`SkyCoord`** objects. One suggested way is to use with keywords appropriate for the frame (e.g., `ra` and `dec` for equatorial systems):

```
>>> from astropy import units as u
>>> ICRS(ra=1.1*u.deg, dec=2.2*u.deg)
<ICRS Coordinate: (ra, dec) in deg
    (1.1, 2.2)>
>>> FK5(ra=1.1*u.deg, dec=2.2*u.deg, equinox='J1975')
<FK5 Coordinate (equinox=J1975.000): (ra, dec) in deg
    (1.1, 2.2)>
```

These same attributes can be used to access the data in the frames as **`Angle`** objects (or **`Angle`** subclasses):

```
>>> coo = ICRS(ra=1.1*u.deg, dec=2.2*u.deg)
>>> coo.ra
<Longitude 1.1 deg>
>>> coo.ra.value
1.1
>>> coo.ra.to(u.hourangle)
<Longitude 0.07333333 hourangle>
```

You can use the `representation_type` attribute in conjunction with the `representation_component_names` attribute to figure out what keywords are accepted by a particular class object. The former will be the representation class in which the system is expressed (e.g., spherical for equatorial frames), and the latter will be a dictionary mapping names for that frame to the attribute name on the representation class:

```
>>> import astropy.units as u
>>> icrs = ICRS(1*u.deg, 2*u.deg)
>>> icrs.representation_type
<class 'astropy.coordinates.representation.SphericalRepresentation'>
>>> icrs.representation_component_names
OrderedDict([('ra', 'lon'), ('dec', 'lat'), ('distance',
'distance')])
```

You can get the data in a different representation if needed:

```
>>> icrs.represent_as('cartesian')
<CartesianRepresentation (x, y, z) [dimensionless]
     (0.99923861, 0.01744177, 0.0348995)>
```

Note

In previous versions of Astropy, both the frame attribute and the argument to frame classes that are now named `representation_type` used to be simply `representation`. The name of this attribute/argument is confusing as it points to the representation *class*, not the object containing the underlying frame data (which is accessed via the frame attribute `.data`). To clarify, we have renamed `representation` to `representation_type`. In this version 3.0, we have only changed the references to this attribute in the documentation. In the next major version, we will issue a deprecation warning. In two major versions, we will remove the `.representation` attribute and `representation=` argument.

The representation of the coordinate object can also be changed directly, as shown below. This does *nothing* to the object internal data which stores the coordinate values, but it changes the external view of that data in two ways: (1)

the object prints itself in accord with the new representation, and (2) the available attributes change to match those of the new representation (e.g., from `ra, dec, distance` to `x, y, z`). Setting the `representation_type` thus changes a *property* of the object (how it appears) without changing the intrinsic object itself which represents a point in 3D space.:

```
>>> from astropy.coordinates import CartesianRepresentation
>>> icrs.representation_type = CartesianRepresentation
>>> icrs
<ICRS Coordinate: (x, y, z) [dimensionless]
    (0.99923861, 0.01744177, 0.0348995)>
>>> icrs.x
<Quantity 0.99923861>
```

The representation can also be set at the time of creating a coordinate and affects the set of keywords used to supply the coordinate data. For example, to create a coordinate with Cartesian data do:

```
>>> ICRS(x=1*u.kpc, y=2*u.kpc, z=3*u.kpc,
representation_type='cartesian')
<ICRS Coordinate: (x, y, z) in kpc
    (1., 2., 3.)>
```

For more information about the use of representations in coordinates see the Representations section, and for details about the representations themselves see Using and Designing Coordinate Representations.

There are two other ways to create frame classes with coordinates. A representation class can be passed in directly at creation, along with any frame attributes required:

```
>>> from astropy.coordinates import SphericalRepresentation
>>> rep = SphericalRepresentation(lon=1.1*u.deg, lat=2.2*u.deg,
distance=3.3*u.kpc)
>>> FK5(rep, equinox='J1975')
<FK5 Coordinate (equinox=J1975.000): (ra, dec, distance) in (deg,
deg, kpc)
    (1.1, 2.2, 3.3)>
```

A final way is to create a frame object from an already existing frame (either one with or without data), using the `realize_frame` method. This will yield a frame with the same attributes, but new data:

```
>>> f1 = FK5(equinox='J1975')
>>> f1
<FK5 Frame (equinox=J1975.000)>
```

```
>>> rep = SphericalRepresentation(lon=1.1*u.deg, lat=2.2*u.deg,
distance=3.3*u.kpc)
>>> f1.realize_frame(rep)
<FK5 Coordinate (equinox=J1975.000): (ra, dec, distance) in (deg,
deg, kpc)
    (1.1, 2.2, 3.3)>
```

You can check if a frame object has data using the `has_data` attribute, and if it is present, it can be accessed from the `data` attribute:

```
>>> ICRS().has_data
False
>>> cooi = ICRS(ra=1.1*u.deg, dec=2.2*u.deg)
>>> cooi.has_data
True
>>> cooi.data
<UnitSphericalRepresentation (lon, lat) in deg
    (1.1, 2.2)>
```

All of the above methods can also accept array data (in the form of class:**Quantity**, or other Python sequences) to create arrays of coordinates:

```
>>> ICRS(ra=[1.5, 2.5]*u.deg, dec=[3.5, 4.5]*u.deg)
<ICRS Coordinate: (ra, dec) in deg
    [(1.5, 3.5), (2.5, 4.5)]>
```

If you pass in mixed arrays and scalars, the arrays will be broadcast over the scalars appropriately:

```
>>> ICRS(ra=[1.5, 2.5]*u.deg, dec=[3.5, 4.5]*u.deg, distance=5*u.kpc)
<ICRS Coordinate: (ra, dec, distance) in (deg, deg, kpc)
    [(1.5, 3.5, 5.), (2.5, 4.5, 5.)]>
```

Similar broadcasting happens if you transform to another frame. For example:

```
>>> import numpy as np
>>> from astropy.coordinates import EarthLocation, AltAz
>>> coo = ICRS(ra=180.*u.deg, dec=51.477811*u.deg)
>>> lf = AltAz(location=EarthLocation.of_site('greenwich'),
...             obstime=['2012-03-21T00:00:00',
'2012-06-21T00:00:00'])
>>> lcoo = coo.transform_to(lf)  # this can load finals2000A.all
>>> lcoo
<AltAz Coordinate (obstime=['2012-03-21T00:00:00.000'
'2012-06-21T00:00:00.000'], location=(3980608.9024681724,
-102.47522910648239, 4966861.273100675) m, pressure=0.0 hPa,
temperature=0.0 deg_C, relative_humidity=0.0, obswl=1.0 micron): (az,
alt) in deg
    [( 94.71264944, 89.21424252), (307.69488825, 37.98077771)]>
```

Above, the shapes — `()` for `coo` and `(2,)` for `lf` — were broadcast against each other. If you wish to determine the positions for a set of coordinates, you will need to make sure that the shapes allow this:

```
>>> coo2 = ICRS(ra=[180., 225., 270.]*u.deg, dec=[51.5, 0.,
51.5]*u.deg)
>>> coo2.transform_to(lf)
Traceback (most recent call last):
...
ValueError: operands could not be broadcast together...
>>> coo2.shape
(3,)
>>> lf.shape
(2,)
>>> lf2 = lf[:, np.newaxis]
>>> lf2.shape
(2, 1)
>>> coo2.transform_to(lf2)
<AltAz Coordinate (obstime=[['2012-03-21T00:00:00.000'
'2012-03-21T00:00:00.000'
  '2012-03-21T00:00:00.000']
 ['2012-06-21T00:00:00.000' '2012-06-21T00:00:00.000'
  '2012-06-21T00:00:00.000']], location=(3980608.90246817,
-102.47522911, 4966861.27310068) m, pressure=0.0 hPa, temperature=0.0
deg_C, relative_humidity=0.0, obswl=1.0 micron): (az, alt) in deg
    [[( 93.09845185, 89.21613128), (126.85789663, 25.4660055 ),
      ( 51.37993234, 37.18532527)],
     [(307.71713698, 37.99437658), (231.3740787 , 26.36768329),
      ( 85.4218724 , 89.69297998)]]>
```

Note

Frames without data have a `shape` that is determined by their frame attributes. For frames with data, the `shape` always is that of the data; any non-scalar attributes are broadcast to have matching shapes (as can be seen for `obstime` in the last line above).

Coordinate values in a array-valued frame object can be modified in-place (added in astropy 4.1). This requires that the new values be set from an another frame object that is equivalent in all ways except for the actual coordinate data values. In this way, no frame transformations are required and the item setting operation is extremely robust.

To modify an array of coordinates use the same syntax for a numpy array:

```
>>> coo1 = ICRS([1, 2] * u.deg, [3, 4] * u.deg)
>>> coo2 = ICRS(10 * u.deg, 20 * u.deg)
>>> coo1[0] = coo2
>>> coo1
<ICRS Coordinate: (ra, dec) in deg
    [(10., 20.), ( 2.,  4.)]>
```

This method is relatively slow because it requires setting from an existing frame object and it performs extensive validation to ensure that the operation is valid. For some applications it may be necessary to take a different lower-level approach which is described in the section Fast In-Place Modification of Coordinates.

Warning

You may be tempted to try an apparently obvious way of modifying a frame object in place by updating the component attributes directly, for example `coo1.ra[1] = 40 * u.deg`. However, while this will *appear* to give a correct result it does not actually modify the underlying representation data. This is related to the current implementation of performance-based caching. The current cache implementation is similarly unable to handle in-place changes to the representation (`.data`) or frame attributes such as `.obstime`.

Transforming between Frames

To transform a frame object with data into another frame, use the `transform_to` method of an object, and provide it the frame you wish to transform to. This frame should be a frame object (with or without coordinate data). If you wish to use all default frame attributes, you can instantiate the frame class with no arguments (i.e., empty parentheses):

```
>>> cooi = ICRS(1.5*u.deg, 2.5*u.deg)
>>> cooi.transform_to(FK5())
<FK5 Coordinate (equinox=J2000.000): (ra, dec) in deg
    (1.50000661, 2.50000238)>
>>> cooi.transform_to(FK5(equinox='J1975'))
<FK5 Coordinate (equinox=J1975.000): (ra, dec) in deg
    (1.17960348, 2.36085321)>
```

The Reference/API includes a list of all of the frames built into

astropy.coordinates, as well as the defined transformations between them. Any transformation that has a valid path, even if it passes through other frames, can be transformed too. To programmatically check for or manipulate transformations, see the **TransformGraph** documentation.

Defining a New Frame

Implementing a new frame class that connects to the `astropy.coordinates` infrastructure can be done by subclassing **BaseCoordinateFrame**. Some guidance and examples are given below, but detailed instructions for creating new frames are given in the docstring of **BaseCoordinateFrame**.

All frame classes must specify a default representation for the coordinate positions by, at minimum, defining a `default_representation` class attribute (see Using and Designing Coordinate Representations for more information about the supported `Representation` objects).

Examples

To create a new frame that, by default, expects to receive its coordinate data in spherical coordinates, we would create a subclass as follows:

```
>>> from astropy.coordinates import BaseCoordinateFrame
>>> import astropy.coordinates.representation as r
>>> class MyFrame1(BaseCoordinateFrame):
...     # Specify how coordinate values are represented when
outputted
...     default_representation = r.SphericalRepresentation
```

Already, this is a valid frame class:

```
>>> fr = MyFrame1(1*u.deg, 2*u.deg)
>>> fr
<MyFrame1 Coordinate: (lon, lat) in deg
    (1., 2.)>
>>> fr.lon
<Longitude 1. deg>
```

However, as we have defined it above, (1) the coordinate component names will be the same as used in the specified `default_representation` (in this case, `lon`, `lat`, and `distance` for longitude, latitude, and distance, respectively), (2) this frame does not have any additional attributes or metadata, (3) this frame does not support transformations to any other coordinate frame, and (4) this frame does not support velocity data. We can address each of these points by seeing some other ways of customizing frame subclasses.

Customizing Frame Component Names

First, as mentioned in the point (1) above, some frame classes have special names for their components. For example, the `ICRS` frame and other equatorial frame classes often use “Right Ascension” or “RA” in place of longitude, and “Declination” or “Dec.” in place of latitude. These component name overrides, which change the frame component name defaults taken from the `Representation` classes, are defined by specifying a set of `RepresentationMapping` instances (one per component) as a part of defining an additional class attribute on a frame class: `frame_specific_representation_info`. This attribute must be a dictionary, and the keys should be either `Representation` or `Differential` classes (see below for a discussion about customizing behavior for velocity components, which is done with the `Differential` classes). Using our example frame implemented above, we can customize it to use the names “R” and “D” instead of “lon” and “lat”:

```
>>> from astropy.coordinates import RepresentationMapping
>>> class MyFrame2(BaseCoordinateFrame):
...     # Specify how coordinate values are represented when
outputted
...     default_representation = r.SphericalRepresentation
...
...     # Override component names (e.g., "ra" instead of "lon")
...     frame_specific_representation_info = {
...         r.SphericalRepresentation: [RepresentationMapping('lon',
'R'),
...                                     RepresentationMapping('lat',
'D')]
...     }
```

With this frame, we can now use the names `R` and `D` to access the frame data:

```
>>> fr = MyFrame2(3*u.deg, 4*u.deg)
>>> fr
<MyFrame2 Coordinate: (R, D) in deg
    (3., 4.)>
>>> fr.R
<Longitude 3. deg>
```

We can specify name mappings for any `Representation` class in `astropy.coordinates` to change the default component names. For example, the **Galactic** frame uses the standard longitude and latitude names “l” and “b” when used with a **SphericalRepresentation**, but uses the component names “x”, “y”, and “z” when the representation is changed to a **CartesianRepresentation**. With our example above, we could add an additional set of mappings to override the Cartesian component names to be “a”, “b”, and “c” instead of the default “x”, “y”, and “z”:

```
>>> class MyFrame3(BaseCoordinateFrame):
...     # Specify how coordinate values are represented when 
outputted
...     default_representation = r.SphericalRepresentation
...
...     # Override component names (e.g., "ra" instead of "lon")
...     frame_specific_representation_info = {
...         r.SphericalRepresentation: [RepresentationMapping('lon', 
'R'),
...                                     RepresentationMapping('lat', 
'D')],
...         r.CartesianRepresentation: [RepresentationMapping('x', 
'a'),
...                                     RepresentationMapping('y', 
'b'),
...                                     RepresentationMapping('z', 
'c')]
...     }
```

For any **RepresentationMapping**, you can also specify a default unit for the component by setting the `defaultunit` keyword argument.

Defining Frame Attributes

Second, as indicated by the point (2) in the introduction above, it is often useful for coordinate frames to allow specifying frame “attributes” that may specify additional data or parameters needed in order to fully specify transformations between a given frame and some other frame. For example, the **FK5** frame allows specifying an `equinox` that helps define the transformation between **FK5** and the **ICRS** frame. Frame attributes are defined by creating class attributes that are instances of **Attribute** or its subclasses (e.g., **TimeAttribute**, **QuantityAttribute**, etc.). If attributes are defined using these classes, there is often no need to define an `__init__` function, as the initializer in **BaseCoordinateFrame** will probably behave in the way you want. Let us now modify the above toy frame class implementation to add two frame attributes:

```
>>> from astropy.coordinates import TimeAttribute, QuantityAttribute
>>> class MyFrame4(BaseCoordinateFrame):
...     # Specify how coordinate values are represented when
outputted
...     default_representation = r.SphericalRepresentation
...
...     # Override component names (e.g., "ra" instead of "lon")
...     frame_specific_representation_info = {
...         r.SphericalRepresentation: [RepresentationMapping('lon',
'R'),
...                                     RepresentationMapping('lat',
'D')],
...         r.CartesianRepresentation: [RepresentationMapping('x',
'a'),
...                                     RepresentationMapping('y',
'b'),
...                                     RepresentationMapping('z',
'c')]
...     }
...
...     # Specify frame attributes required to fully specify the
frame
...     time = TimeAttribute(default='B1950')
...     orientation = QuantityAttribute(default=42*u.deg)
```

Without specifying an initializer, defining these attributes tells the `BaseCoordinateFrame` what to expect in terms of additional arguments passed in to our subclass initializer. For example, when defining a frame instance with our subclass, we can now optionally specify values for these attributes:

```
>>> fr = MyFrame4(R=1*u.deg, D=2*u.deg, orientation=21*u.deg)
>>> fr
<MyFrame4 Coordinate (time=B1950.000, orientation=21.0 deg): (R, D)
in deg
    (1., 2.)>
```

Note that we specified both frame attributes with default values, so they are optional arguments to the frame initializer. Note also that the frame attributes now appear in the `repr` of the frame instance above. As a bonus, for most of the `Attribute` subclasses, even without defining an initializer, attributes specified as arguments will be validated. For example, arguments passed in to `QuantityAttribute` attributes will be checked that they have valid and compatible units with the expected attribute units. Using our frame example above, which expects an `orientation` with angular units, passing in a time results in an error:

```
>>> MyFrame4(R=1*u.deg, D=2*u.deg, orientation=55*u.microyear)
Traceback (most recent call last):
...
UnitConversionError: 'uyr' (time) and 'deg' (angle) are not
convertible
```

When defining frame attributes, you do not always have to specify a default value as long as the `Attribute` subclass is able to validate the input. For example, with the above frame, if the `orientation` does not require a default value but we still want to enforce it to have angular units, we could instead define it as:

```
orientation = QuantityAttribute(unit=u.deg)
```

In the above case, if `orientation` is not specified when a new frame instance is created, its value will be `None`: Note that it is up to the frame classes and transformation function implementations to define how to handle a `None` value. In most cases `None` should signify a special case like “use a different frame attribute for this value” or similar.

Customizing Display of Attributes

While the default `repr` for coordinate frames is suitable for most cases, you may want to customize how frame attributes are displayed in certain cases. To do this you can define a method named `_astropy_repr_in_frame`. This method should be defined on the object that is set to the frame attribute itself, **not** the `Attribute` descriptor.

Example

As an example of method `_astropy_repr_in_frame`, say you have an object `Spam` which you have as an attribute of your frame:

```
>>> class Spam:
...     def _astropy_repr_in_frame(self):
...         return "<A can of Spam>"
```

If your frame has this class as an attribute:

```
>>> from astropy.coordinates import Attribute
>>> class Egg(BaseCoordinateFrame):
...     can = Attribute(default=Spam())
```

When it is displayed by the frame it will use the result of `_astropy_repr_in_frame`:

```
>>> Egg()
<Egg Frame (can=<A can of Spam>)>
```

Defining Transformations between Frames

As indicated by the point (3) in the introduction above, a frame class on its own is likely not very useful until transformations are defined between it and other coordinate frame classes. The key concept for defining transformations in `astropy.coordinates` is the "frame transform graph" (in the "graph theory" sense, not "plot"), which stores all of the transformations between the built-in frames, as well as tools for finding the shortest paths through this graph to transform from any frame to any other by composing the transformations. The power behind this concept is available to user-created frames as well, meaning that once you define even one transform from your frame to any frame in the graph, coordinates defined in your frame can be transformed to *any* other frame in the graph. The "frame transform graph" is available in code as `astropy.coordinates.frame_transform_graph`, which is an instance of the **`TransformGraph`** class.

The transformations themselves are represented as **`CoordinateTransform`** objects or their subclasses. The useful subclasses/types of transformations are:

- **`FunctionTransform`**

  A transform that is defined as a function that takes a frame object of one frame class and returns an object of another class.

- **`AffineTransform`**

  A transformation that includes a linear matrix operation and a translation (vector offset). These transformations are defined by a 3x3 matrix and a 3-vector for the offset (supplied as a Cartesian representation). The transformation is applied to the Cartesian representation of one frame and transforms into the Cartesian representation of the target frame.

- **`StaticMatrixTransform`**
- **`DynamicMatrixTransform`**

  The matrix transforms are **`AffineTransform`** transformations without a translation (i.e., only a rotation). The static version is for the case where the matrix is independent of the frame attributes (e.g., the ICRS->FK5 transformation, because ICRS has no frame attributes). The dynamic case is for transformations where the transformation matrix depends on the frame attributes of either the to or from frame.

Generally, it is not necessary to use these classes directly. Instead, use methods on the `frame_transform_graph` that can be used as function decorators. Define functions that either do the actual transformation (for **`FunctionTransform`**), or that compute the necessary transformation matrices to transform. Then decorate the functions to register these transformations with the frame transform graph:

```
from astropy.coordinates import frame_transform_graph

@frame_transform_graph.transform(DynamicMatrixTransform, ICRS, FK5)
def icrs_to_fk5(icrscoord, fk5frame):
    ...

@frame_transform_graph.transform(DynamicMatrixTransform, FK5, ICRS)
def fk5_to_icrs(fk5coord, icrsframe):
    ...
```

If the transformation to your coordinate frame of interest is not representable by a matrix operation, you can also specify a function to do the actual transformation, and pass the **`FunctionTransform`** class to the transform graph decorator instead:

```
@frame_transform_graph.transform(FunctionTransform, FK4NoETerms, FK4)
def fk4_no_e_to_fk4(fk4noecoord, fk4frame):
    ...
```

Furthermore, the `frame_transform_graph` does some caching and optimization to speed up transformations after the first attempt to go from one frame to another, and shortcuts steps where relevant (for example, combining multiple static matrix transforms into a single matrix). Hence, in general, it is better to define whatever are the most natural transformations for a user-defined frame, rather than worrying about optimizing or caching a transformation to speed up the process.

For a demonstration of how to define transformation functions that also work for transforming velocity components, see Transforming Frames with Velocities.

Supporting Velocity Data in Frames

As alluded to by point (4) in the introduction above, the examples we have seen above mostly deal with customizing frame behavior for positional information. (For some context about how velocities are handled in `astropy.coordinates`, it may be useful to read the overview: Creating Frame Objects with Velocity Data.)

When defining a frame class, it is also possible to set a `default_differential` (analogous to `default_representation`), and to customize how velocity data components are named. Expanding on our custom frame example above, we can use **RepresentationMapping** to override `Differential` component names. The default `Differential` components are typically named after the corresponding `Representation` component, preceded by `d_`. So, for example, the longitude `Differential` component is, by default, `d_lon`. However, there are some defaults to be aware of. Here, if we set the default `Differential` class to also be Spherical, it will implement a set of default "nicer" names for the velocity components, mapping `pm_R` to `d_lon`, `pm_D` to `d_lat`, and `radial_velocity` to `d_distance` (taking the previously overridden longitude and latitude component names):

```
>>> class MyFrame4WithVelocity(BaseCoordinateFrame):
...     # Specify how coordinate values are represented when
outputted
...     default_representation = r.SphericalRepresentation
...     default_differential = r.SphericalDifferential
...
...     # Override component names (e.g., "ra" instead of "lon")
...     frame_specific_representation_info = {
...         r.SphericalRepresentation: [RepresentationMapping('lon',
'R'),
...                                     RepresentationMapping('lat',
'D')],
...         r.CartesianRepresentation: [RepresentationMapping('x',
'a'),
...                                     RepresentationMapping('y',
'b'),
...                                     RepresentationMapping('z',
'c')]
...     }
>>> fr = MyFrame4WithVelocity(R=1*u.deg, D=2*u.deg,
...                           pm_R=3*u.mas/u.yr, pm_D=4*u.mas/u.yr)
>>> fr
<MyFrame4WithVelocity Coordinate: (R, D) in deg
    (1., 2.)
(pm_R, pm_D) in mas / yr
    (3., 4.)>
```

If you want to override the default "nicer" names, you can specify a new key in the `frame_specific_representation_info` for any of the `Differential` classes, for example:

```
>>> class MyFrame4WithVelocity2(BaseCoordinateFrame):
...     # Specify how coordinate values are represented when
outputted
...     default_representation = r.SphericalRepresentation
...     default_differential = r.SphericalDifferential
...
...     # Override component names (e.g., "ra" instead of "lon")
...     frame_specific_representation_info = {
...         r.SphericalRepresentation: [RepresentationMapping('lon',
'R'),
...                                     RepresentationMapping('lat',
'D')],
...         r.CartesianRepresentation: [RepresentationMapping('x',
'a'),
...                                     RepresentationMapping('y',
'b'),
...                                     RepresentationMapping('z',
'c')],
...         r.SphericalDifferential: [RepresentationMapping('d_lon',
'pm1'),
...                                   RepresentationMapping('d_lat',
'pm2'),
...
RepresentationMapping('d_distance', 'rv')]
...     }
>>> fr = MyFrame4WithVelocity2(R=1*u.deg, D=2*u.deg,
...                              pm1=3*u.mas/u.yr, pm2=4*u.mas/u.yr)
>>> fr
<MyFrame4WithVelocity2 Coordinate: (R, D) in deg
    (1., 2.)
(pm1, pm2) in mas / yr
    (3., 4.)>
```

Final Notes

You can also define arbitrary methods for any added functionality you want your frame to have that is unique to that frame. These methods will be available in any **SkyCoord** that is created using your user-defined frame.

For examples of defining frame classes, the first place to look is at the source code for the frames that are included in `astropy` (available at `astropy.coordinates.builtin_frames`). These are not special-cased, but rather use all of the same API and features available to user-created frames.

Examples:

See also Create a new coordinate class (for the Sagittarius stream) for a more annotated example of defining a new coordinate frame.

Working with Velocities in Astropy Coordinates

Using Velocities with `SkyCoord`

The best way to start getting a coordinate object with velocities is to use the **SkyCoord** interface.

Examples

A **SkyCoord** to represent a star with a measured radial velocity but unknown proper motion and distance could be created as:

```
>>> from astropy.coordinates import SkyCoord
>>> import astropy.units as u
>>> sc = SkyCoord(1*u.deg, 2*u.deg, radial_velocity=20*u.km/u.s)
>>> sc
<SkyCoord (ICRS): (ra, dec) in deg
    (1., 2.)
 (radial_velocity) in km / s
    (20.,)>
>>> sc.radial_velocity
<Quantity 20.0 km / s>
```

SkyCoord objects created in this manner follow all of the same transformation rules and will correctly update their velocities when transformed to other frames. For example, to determine proper motions in Galactic coordinates for a star with proper motions measured in ICRS:

```
>>> sc = SkyCoord(1*u.deg, 2*u.deg, pm_ra_cosdec=.2*u.mas/u.yr,
pm_dec=.1*u.mas/u.yr)
>>> sc.galactic
<SkyCoord (Galactic): (l, b) in deg
  ( 99.63785528, -58.70969293)
(pm_l_cosb, pm_b) in mas / yr
  ( 0.22240398,  0.02316181)>
```

For more details on valid operations and limitations of velocity support in **astropy.coordinates** (particularly the current accuracy limitations), see the more detailed discussions below of velocity support in the lower-level frame objects. All these same rules apply for **SkyCoord** objects, as they are built directly on top of the frame classes' velocity functionality detailed here.

Creating Frame Objects with Velocity Data

The coordinate frame classes support storing and transforming velocity data (alongside the positional coordinate data). Similar to the positional data that use the `Representation` classes to abstract away the particular representation and allow re-representing from (e.g., Cartesian to Spherical), the velocity data makes use of `Differential` classes to do the same. (For more information about the differential classes, see Differentials and Derivatives of Representations.) Also like the positional data, the names of the differential (velocity) components depend on the particular coordinate frame.

Most frames expect velocity data in the form of two proper motion components and/or a radial velocity because the default differential for most frames is the **`SphericalCosLatDifferential`** class. When supported, the proper motion components all begin with `pm_` and, by default, the longitudinal component is expected to already include the `cos(latitude)` term. For example, the proper motion components for the `ICRS` frame are (`pm_ra_cosdec`, `pm_dec`).

Examples

To create frame objects with velocity data in the form of proper motion components:

```
>>> from astropy.coordinates import ICRS
>>> ICRS(ra=8.67*u.degree, dec=53.09*u.degree,
...      pm_ra_cosdec=4.8*u.mas/u.yr, pm_dec=-15.16*u.mas/u.yr)
<ICRS Coordinate: (ra, dec) in deg
    (8.67, 53.09)
 (pm_ra_cosdec, pm_dec) in mas / yr
    (4.8, -15.16)>
>>> ICRS(ra=8.67*u.degree, dec=53.09*u.degree,
...      pm_ra_cosdec=4.8*u.mas/u.yr, pm_dec=-15.16*u.mas/u.yr,
...      radial_velocity=23.42*u.km/u.s)
<ICRS Coordinate: (ra, dec) in deg
    (8.67, 53.09)
 (pm_ra_cosdec, pm_dec, radial_velocity) in (mas / yr, mas / yr, km /
s)
    (4.8, -15.16, 23.42)>
```

For proper motion components in the `Galactic` frame, the names track the longitude and latitude names:

```
>>> from astropy.coordinates import Galactic
>>> Galactic(l=11.23*u.degree, b=58.13*u.degree,
...          pm_l_cosb=21.34*u.mas/u.yr, pm_b=-55.89*u.mas/u.yr)
```

```
<Galactic Coordinate: (l, b) in deg
    (11.23, 58.13)
 (pm_l_cosb, pm_b) in mas / yr
    (21.34, -55.89)>
```

Like the positional data, velocity data must be passed in as **Quantity** objects.

The expected differential class can be changed to control the argument names that the frame expects. By default the proper motion components are expected to contain the `cos(latitude)`, but this can be changed by specifying the **SphericalDifferential** class (instead of the default **SphericalCosLatDifferential**):

```
>>> from astropy.coordinates import SphericalDifferential
>>> Galactic(l=11.23*u.degree, b=58.13*u.degree,
...          pm_l=21.34*u.mas/u.yr, pm_b=-55.89*u.mas/u.yr,
...          differential_type=SphericalDifferential)
<Galactic Coordinate: (l, b) in deg
    (11.23, 58.13)
 (pm_l, pm_b) in mas / yr
    (21.34, -55.89)>
```

This works in parallel to specifying the expected representation class, as long as the differential class is compatible with the representation. For example, to specify all coordinate and velocity components in Cartesian:

```
>>> from astropy.coordinates import (CartesianRepresentation,
...                                  CartesianDifferential)
>>> Galactic(u=103*u.pc, v=-11*u.pc, w=93.*u.pc,
...          U=31*u.km/u.s, V=-10*u.km/u.s, W=75*u.km/u.s,
...          representation_type=CartesianRepresentation,
...          differential_type=CartesianDifferential)
<Galactic Coordinate: (u, v, w) in pc
    (103., -11., 93.)
 (U, V, W) in km / s
    (31., -10., 75.)>
```

Note that the `Galactic` frame has special, standard names for Cartesian position and velocity components. For other frames, these are just `x,y,z` and `v_x,v_y,v_z`:

```
>>> ICRS(x=103*u.pc, y=-11*u.pc, z=93.*u.pc,
...      v_x=31*u.km/u.s, v_y=-10*u.km/u.s, v_z=75*u.km/u.s,
...      representation_type=CartesianRepresentation,
...      differential_type=CartesianDifferential)
<ICRS Coordinate: (x, y, z) in pc
```

```
    (103., -11., 93.)
 (v_x, v_y, v_z) in km / s
    (31., -10., 75.)>
```

For any frame with velocity data with any representation, there are also shorthands that provide easier access to the underlying velocity data in commonly needed formats. With any frame object with 3D velocity data, the 3D Cartesian velocity can be accessed with:

```
>>> icrs = ICRS(ra=8.67*u.degree, dec=53.09*u.degree,
...             distance=171*u.pc,
...             pm_ra_cosdec=4.8*u.mas/u.yr, pm_dec=-
15.16*u.mas/u.yr,
...             radial_velocity=23.42*u.km/u.s)
>>> icrs.velocity
<CartesianDifferential (d_x, d_y, d_z) in km / s
    ( 23.03160789,  7.44794505,  11.34587732)>
```

There are also shorthands for retrieving a single **Quantity** object that contains the two-dimensional proper motion data, and for retrieving the radial (line-of-sight) velocity:

```
>>> icrs.proper_motion
<Quantity [  4.8 ,-15.16] mas / yr>
>>> icrs.radial_velocity
<Quantity 23.42 km / s>
```

Adding Velocities to Existing Frame Objects

Another use case similar to the above comes up when you have an existing frame object (or **SkyCoord**) and want an object with the same position but with velocities added. The most conceptually direct way to do this is to use the differential objects along with **realize_frame**.

Examples
The following snippet accomplishes a well-defined case where the desired velocities are known in the Cartesian representation. To add the velocities to the existing frame using **realize_frame**:

```
>>> icrs = ICRS(1*u.deg, 2*u.deg, distance=3*u.kpc)
>>> icrs
<ICRS Coordinate: (ra, dec, distance) in (deg, deg, kpc)
    (1., 2., 3.)>
```

```
>>> vel_to_add = CartesianDifferential(4*u.km/u.s, 5*u.km/u.s,
6*u.km/u.s)
>>> newdata = icrs.data.to_cartesian().with_differentials(vel_to_add)
>>> icrs.realize_frame(newdata)
<ICRS Coordinate: (ra, dec, distance) in (deg, deg, kpc)
    (1., 2., 3.)
 (pm_ra_cosdec, pm_dec, radial_velocity) in (mas / yr, mas / yr, km /
s)
    (0.34662023, 0.41161335, 4.29356031)>
```

A similar mechanism can also be used to add velocities even if full 3D coordinates are not available (e.g., for a radial velocity observation of an object where the distance is unknown). However, it requires a slightly different way of specifying the differentials because of the lack of explicit unit information:

```
>>> from astropy.coordinates import RadialDifferential
>>> icrs_no_distance = ICRS(1*u.deg, 2*u.deg)
>>> icrs_no_distance
<ICRS Coordinate: (ra, dec) in deg
    (1., 2.)>
>>> rv_to_add = RadialDifferential(500*u.km/u.s)
>>> data_with_rv =
icrs_no_distance.data.with_differentials({'s':rv_to_add})
>>> icrs_no_distance.realize_frame(data_with_rv)
<ICRS Coordinate: (ra, dec) in deg
    (1., 2.)
 (radial_velocity) in km / s
    (500.,)>
```

Which we can see yields an object identical to what you get when you specify a radial velocity when you create the object:

```
>>> ICRS(1*u.deg, 2*u.deg, radial_velocity=500*u.km/u.s)
<ICRS Coordinate: (ra, dec) in deg
    (1., 2.)
 (radial_velocity) in km / s
    (500.,)>
```

Transforming Frames with Velocities

Transforming coordinate frame instances that contain velocity data to a different frame (which may involve both position and velocity transformations) is done exactly the same way as transforming position-only frame instances.

Example

To transform a coordinate frame that contains velocity data:

```
>>> from astropy.coordinates import Galactic
>>> icrs = ICRS(ra=8.67*u.degree, dec=53.09*u.degree,
...             pm_ra_cosdec=4.8*u.mas/u.yr, pm_dec=-
15.16*u.mas/u.yr)
>>> icrs.transform_to(Galactic())
<Galactic Coordinate: (l, b) in deg
    (120.38084191, -9.69872044)
 (pm_l_cosb, pm_b) in mas / yr
    (3.78957965, -15.44359693)>
```

However, the details of how the velocity components are transformed depends on the particular set of transforms required to get from the starting frame to the desired frame (i.e., the path taken through the frame transform graph). If all frames in the chain of transformations are transformed to each other via **BaseAffineTransform** subclasses (i.e., are matrix transformations or affine transformations), then the transformations can be applied explicitly to the velocity data. If this is not the case, the velocity transformation is computed numerically by finite-differencing the positional transformation. See the subsections below for more details about these two methods.

Affine Transformations

Frame transformations that involve a rotation and/or an origin shift and/or a velocity offset are implemented as affine transformations using the **BaseAffineTransform** subclasses: **StaticMatrixTransform**, **DynamicMatrixTransform**, and **AffineTransform**.

Matrix-only transformations (e.g., rotations such as **ICRS** to **Galactic**) can be performed on proper-motion-only data or full-space, 3D velocities.

Examples

To perform a matrix-only transformation:

```
>>> icrs = ICRS(ra=8.67*u.degree, dec=53.09*u.degree,
...             pm_ra_cosdec=4.8*u.mas/u.yr, pm_dec=-
15.16*u.mas/u.yr,
...             radial_velocity=23.42*u.km/u.s)
>>> icrs.transform_to(Galactic())
<Galactic Coordinate: (l, b) in deg
    (120.38084191, -9.69872044)
 (pm_l_cosb, pm_b, radial_velocity) in (mas / yr, mas / yr, km / s)
    (3.78957965, -15.44359693, 23.42)>
```

The same rotation matrix is applied to both the position vector and the velocity vector. Any transformation that involves a velocity offset requires all 3D velocity components (which typically require specifying a distance as well), for example, **ICRS** to **LSR**:

```
>>> from astropy.coordinates import LSR
>>> icrs = ICRS(ra=8.67*u.degree, dec=53.09*u.degree,
...             distance=117*u.pc,
...             pm_ra_cosdec=4.8*u.mas/u.yr, pm_dec=-
15.16*u.mas/u.yr,
...             radial_velocity=23.42*u.km/u.s)
>>> icrs.transform_to(LSR())
<LSR Coordinate (v_bary=(11.1, 12.24, 7.25) km / s): (ra, dec,
distance) in (deg, deg, pc)
    (8.67, 53.09, 117.)
 (pm_ra_cosdec, pm_dec, radial_velocity) in (mas / yr, mas / yr, km /
s)
    (-24.51315607, -2.67935501, 27.07339176)>
```

Finite Difference Transformations

Some frame transformations cannot be expressed as affine transformations. For example, transformations from the **AltAz** frame can include an atmospheric dispersion correction, which is inherently nonlinear. Additionally, some frames are more conveniently implemented as functions, even if they can be cast as affine transformations. For these frames, a finite difference approach to transforming velocities is available. Note that this approach is implemented such that user-defined frames can use it in the same manner (i.e., by defining a transformation of the **FunctionTransformWithFiniteDifference** type).

This finite difference approach actually combines two separate (but important) elements of the transformation:

- Transformation of the *direction* of the velocity vector that already exists in the starting frame. That is, a frame transformation sometimes involves reorienting the coordinate frame (e.g., rotation), and the velocity vector in the new frame must account for this. The finite difference approach models this by moving the position of the starting frame along the velocity vector, and computing this offset in the target frame.
- The “induced” velocity due to motion of the frame *itself*. For example, shifting from a frame centered at the solar system barycenter to one centered on the Earth includes a velocity component due entirely to the Earth’s motion around the barycenter. This is accounted for by computing the location of the starting frame in the target frame at slightly different times, and computing the difference between those. Note that this step depends on assuming

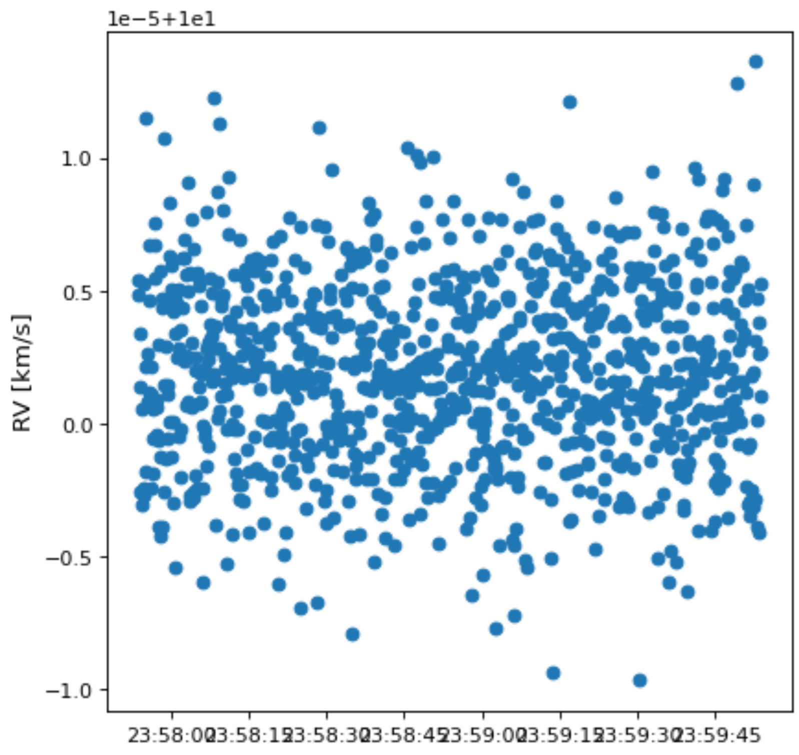
that a particular frame attribute represents a “time” of relevance for the induced velocity. By convention this is typically the `obtime` frame attribute, although it is an option that can be set when defining a finite difference transformation function.

Example

It is important to recognize that the finite difference transformations have inherent limits set by the finite difference algorithm and machine precision. To illustrate this problem, consider the AltAz to GCRS (i.e., geocentric) transformation. Let us try to compute the radial velocity in the GCRS frame for something observed from the Earth at a distance of 100 AU with a radial velocity of 10 km/s:

```
import numpy as np
from matplotlib import pyplot as plt

from astropy import units as u
from astropy.time import Time
from astropy.coordinates import EarthLocation, AltAz, GCRS

time = Time('J2010') + np.linspace(-1,1,1000)*u.min
location = EarthLocation(lon=0*u.deg, lat=45*u.deg)
aa = AltAz(alt=[45]*1000*u.deg, az=90*u.deg, distance=100*u.au,
           radial_velocity=[10]*1000*u.km/u.s,
           location=location, obstime=time)
gcrs = aa.transform_to(GCRS(obstime=time))
plt.plot_date(time.plot_date, gcrs.radial_velocity.to(u.km/u.s))
plt.ylabel('RV [km/s]')
```

(png, svg, pdf)

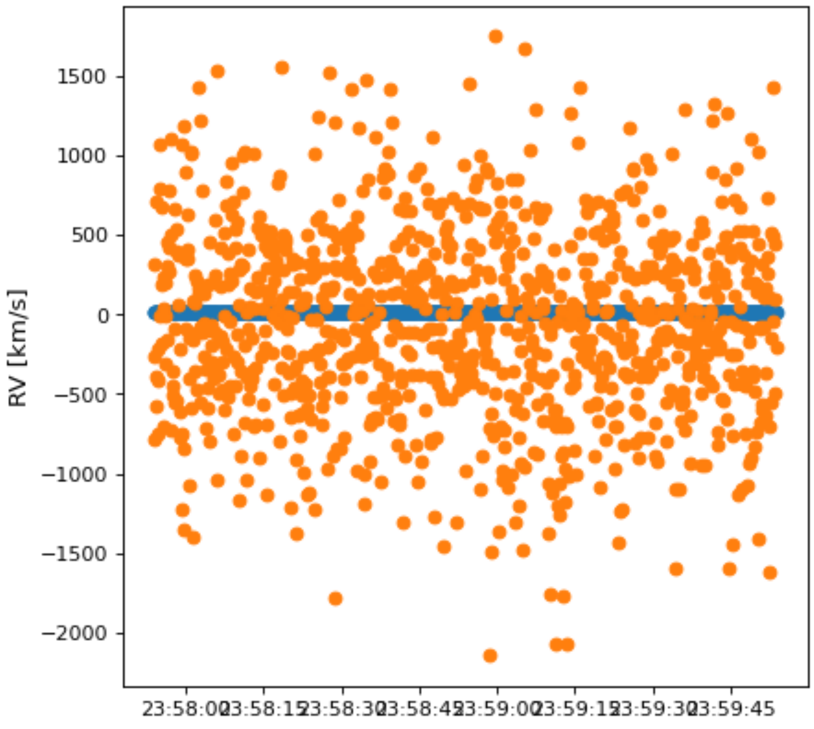


This seems plausible: the radial velocity should indeed be very close to 10 km/s because the frame does not involve a velocity shift.

Now let us consider 100 *kiloparsecs* as the distance. In this case we expect the same: the radial velocity should be essentially the same in both frames:

```
time = Time('J2010') + np.linspace(-1,1,1000)*u.min
location = EarthLocation(lon=0*u.deg, lat=45*u.deg)
aa = AltAz(alt=[45]*1000*u.deg, az=90*u.deg, distance=100*u.kpc,
           radial_velocity=[10]*1000*u.km/u.s,
           location=location, obstime=time)
gcrs = aa.transform_to(GCRS(obstime=time))
plt.plot_date(time.plot_date, gcrs.radial_velocity.to(u.km/u.s))
plt.ylabel('RV [km/s]')
```

(png, svg, pdf)



But this result is nonsense, with values from -1000 to 1000 km/s instead of the ~10 km/s we expected. The root of the problem here is that the machine precision is not sufficient to compute differences of order km over distances of order kiloparsecs. Hence, the straightforward finite difference method will not work for this use case with the default values.

It is possible to override the timestep over which the finite difference occurs. For example:

```
>>> from astropy.coordinates import frame_transform_graph, AltAz,
CIRS
>>> trans = frame_transform_graph.get_transform(AltAz,
CIRS).transforms[0]
>>> trans.finite_difference_dt = 1 * u.year
>>> gcrs = aa.transform_to(GCRS(obstime=time))
>>> trans.finite_difference_dt = 1 * u.second  # return to default
```

In the above example, there is exactly one transformation step from **`AltAz`** to **`GCRS`**. In general, there may be more than one step between two frames, or the single step may perform other transformations internally. One can use the context manager **`impose_finite_difference_dt()`** for the transformation graph to override `finite_difference_dt` for *all* finite-difference transformations on the graph:

```
>>> from astropy.coordinates import frame_transform_graph
>>> with frame_transform_graph.impose_finite_difference_dt(1 *
u.year):
...     gcrs = aa.transform_to(GCRS(obstime=time))
```

But beware that this will *not* help in cases like the above, where the relevant timescales for the velocities are seconds. (The velocity of the Earth relative to a particular direction changes dramatically over the course of one year.)

Future versions of Astropy will improve on this algorithm to make the results more numerically stable and practical for use in these (not unusual) use cases.

Radial Velocity Corrections

Separately from the above, Astropy supports computing barycentric or heliocentric radial velocity corrections. While in the future this may be a high-level convenience function using the framework described above, the current implementation is independent to ensure sufficient accuracy (see Radial Velocity Corrections and the **`radial_velocity_correction`** API docs for details).

Example

This example demonstrates how to compute this correction if observing some object at a known RA and Dec from the Keck observatory at a particular time. If a precision of around 3 m/s is sufficient, the computed correction can then be added to any observed radial velocity to determine the final heliocentric radial velocity:

```
>>> from astropy.time import Time
>>> from astropy.coordinates import SkyCoord, EarthLocation
>>> # keck = EarthLocation.of_site('Keck')  # the easiest way... but
requires internet
>>> keck = EarthLocation.from_geodetic(lat=19.8283*u.deg,
lon=-155.4783*u.deg, height=4160*u.m)
>>> sc = SkyCoord(ra=4.88375*u.deg, dec=35.0436389*u.deg)
>>> barycorr =
sc.radial_velocity_correction(obstime=Time('2016-6-4'),
location=keck)
>>> barycorr.to(u.km/u.s)
<Quantity 20.077135 km / s>
>>> heliocorr = sc.radial_velocity_correction('heliocentric',
obstime=Time('2016-6-4'), location=keck)
>>> heliocorr.to(u.km/u.s)
<Quantity 20.070039 km / s>
```

Note that there are a few different ways to specify the options for the correction (e.g., the location, observation time, etc.). See the **radial_velocity_correction** docs for more information.

Precision of radial_velocity_correction

The correction computed by **radial_velocity_correction** uses the optical approximation $v = zc$ (see Spectral (Doppler) Equivalencies for details). The correction can be added to any observed radial velocity to provide a correction that is accurate to a level of approximately 3 m/s. If you need more precise corrections, there are a number of subtleties of which you must be aware.

The first is that you should always use a barycentric correction, as the barycenter is a fixed point where gravity is constant. Since the heliocenter does not satisfy these conditions, corrections to the heliocenter are only suitable for low precision work. As a result, and to increase speed, the heliocentric correction in **radial_velocity_correction** does not include effects such as the gravitational redshift due to the potential at the Earth’s surface. For these reasons, the barycentric correction in **radial_velocity_correction** should always be used for high precision work.

Other considerations necessary for radial velocity corrections at the cm/s level are outlined in Wright & Eastman (2014). Most important is that the barycentric correction is, strictly speaking, *multiplicative*, so that you should apply it as:

$$v_t = v_m + v_b + \frac{v_b v_m}{c},$$

Where v_t is the true radial velocity, v_m is the measured radial velocity and v_b is the barycentric correction returned by **radial_velocity_correction**. Failure to apply the barycentric correction in this way leads to errors of order 3 m/s.

The barycentric correction in **radial_velocity_correction** is consistent with the IDL implementation of the Wright & Eastmann (2014) paper to a level of 10 mm/s for a source at infinite distance. We do not include the Shapiro delay nor the light travel time correction from equation 28 of that paper. The neglected terms are not important unless you require accuracies of better than 1 cm/s. If you do require that precision, see Wright & Eastmann (2014).

Accounting for Space Motion

The **SkyCoord** object supports updating the position of a source given its space motion and a time at which to evaluate the new position (or a difference between the coordinate’s current time and a new one). This is done using the **apply_space_motion()** method.

Example

First we will create a **SkyCoord** object with a specified `obstime`:

```
>>> import astropy.units as u
>>> from astropy.time import Time
>>> from astropy.coordinates import SkyCoord
>>> c = SkyCoord(l=10*u.degree, b=45*u.degree, distance=100*u.pc,
...              pm_l_cosb=34*u.mas/u.yr, pm_b=-117*u.mas/u.yr,
...              frame='galactic',
...              obstime=Time('1988-12-18 05:11:23.5'))
```

We can now find the position at some other time, taking the space motion into account. We can either specify the time difference between the observation time and the desired time:

```
>>> c.apply_space_motion(dt=10. * u.year)
<SkyCoord (Galactic): (l, b, distance) in (deg, deg, pc)
    ( 10.00013356,  44.999675,  99.99999994)
 (pm_l_cosb, pm_b, radial_velocity) in (mas / yr, mas / yr, km / s)
    ( 33.99980714, -117.00005604,  0.00034117)>
>>> c.apply_space_motion(dt=-10. * u.year)
<SkyCoord (Galactic): (l, b, distance) in (deg, deg, pc)
    ( 9.99986643,  45.000325,  100.00000006)
 (pm_l_cosb, pm_b, radial_velocity) in (mas / yr, mas / yr, km / s)
    ( 34.00019286, -116.99994395, -0.00034117)>
```

Or, we can specify the new time to evaluate the position at:

```
>>> c.apply_space_motion(new_obstime=Time('2017-12-18 01:12:07.3'))
<SkyCoord (Galactic): (l, b, distance) in (deg, deg, pc)
    ( 10.00038732,  44.99905754,  99.99999985)
 (pm_l_cosb, pm_b, radial_velocity) in (mas / yr, mas / yr, km / s)
    ( 33.99944073, -117.00016248,  0.00098937)>
```

If the **SkyCoord** object has no specified radial velocity (RV), the RV is assumed to be 0. The new position of the source is determined assuming the source moves in a straight line with constant velocity in an inertial frame. There are no plans to support more complex evolution (e.g., non-inertial frames or more complex evolution), as that is out of scope for the `astropy` core package (although it may well be in-scope for a variety of affiliated packages).

Example: Use velocity to compute sky position at different epochs

In this example, we will use *Gaia* TGAS astrometry for a nearby star to compute the sky position of the source on the date that the 2MASS survey observed that region of the sky. The TGAS astrometry is provided on the reference epoch J2015.0, whereas the 2MASS survey occurred in the late 1990's. For the star of interest, the proper motion is large enough that there are appreciable differences in the sky position between the two surveys.

After computing the previous position of the source, we will then cross-match the source with the 2MASS catalog to compute *Gaia*-2MASS colors for this object source.

Note

This example requires accessing data from the *Gaia* TGAS and 2MASS catalogs. For convenience and speed below, we have created dictionary objects that contain the data. We retrieved the data using the Astropy affiliated package astroquery using the following queries:

```
import astropy.coordinates as coord
import astropy.units as u
from astroquery.gaia import Gaia
from astroquery.vizier import Vizier

job = Gaia.launch_job("SELECT TOP 1 * FROM gaiadr1.tgas_source \
    WHERE parallax_error < 0.3  AND parallax > 5 AND pmra > 100 \
    ORDER BY random_index")
result_tgas = job.get_results()[0]

c_tgas = coord.SkyCoord(ra=result_tgas['ra'] * u.deg,
                        dec=result_tgas['dec'] * u.deg)
v = Vizier(columns=["**"], catalog="II/246/out")
result_2mass = v.query_region(c, radius=1*u.arcmin)['II/246/out']
```

The TGAS data from relevant columns for this source (see queries in Note above):

```
>>> result_tgas = dict(ra=66.44280212823296,
...                    dec=-69.99366255906372,
...                    parallax=22.764078749733947,
...                    pmra=144.91354358297048,
...                    pmdec=5.445648092997134,
...                    ref_epoch=2015.0,
...                    phot_g_mean_mag=7.657174523348196)
```

The 2MASS data for all sources within 1 arcminute around the above position (see queries in Note above):

```
>>> result_2mass = dict(RAJ2000=[66.421970000000002,
66.433521999999996,
...                              66.420564999999996,
66.485068999999996,
...                              66.467928999999998,
66.440815000000001,
...                              66.440454000000003],
...                        DEJ2000=[-70.003722999999994,
-69.990768000000003,
...                              -69.992255999999998,
-69.994881000000007,
...                              -69.994926000000007,
-69.993613999999994,
...                              -69.990836999999999],
...                        Jmag=[16.35, 13.663, 16.171, 16.184, 16.292,
...                              6.6420002, 12.275],
...                        Hmag=[15.879, 13.955, 15.154, 15.856, 15.642,
...                              6.3660002, 12.185],
...                        Kmag=[15.581, 14.238, 14.622, 15.398, 15.123,
...                              6.2839999, 12.106],
...                        Date=['1998-10-24', '1998-10-24',
'1998-10-24',
...                              '1998-10-24', '1998-10-24',
'1998-10-24',
...                              '1998-10-24'])
```

We will first create a **SkyCoord** object from the information provided in the TGAS catalog. Note that we set the `obstime` of the object to the reference epoch provided by the TGAS catalog (J2015.0):

```
>>> import astropy.units as u
>>> from astropy.coordinates import SkyCoord, Distance
>>> from astropy.time import Time
>>> c = SkyCoord(ra=result_tgas['ra'] * u.deg,
...                  dec=result_tgas['dec'] * u.deg,
...                  distance=Distance(parallax=result_tgas['parallax'] *
u.mas),
...                  pm_ra_cosdec=result_tgas['pmra'] * u.mas/u.yr,
...                  pm_dec=result_tgas['pmdec'] * u.mas/u.yr,
...                  obstime=Time(result_tgas['ref_epoch'],
format='jyear'))
```

We next create a **SkyCoord** object with the sky positions from the 2MASS catalog, and an **Time** object for the date of the 2MASS observations provided in the 2MASS catalog (for the data in this region the observation date is the same, so we take only the 0th value):

```
>>> catalog_2mass = SkyCoord(ra=result_2mass['RAJ2000'] * u.deg,
...                              dec=result_2mass['DEJ2000'] * u.deg)
```

```
>>> epoch_2mass = Time(result_2mass['Date'][0])
```

We can now use the `apply_space_motion()` method to compute the position of the TGAS source at another epoch. This uses the proper motion and parallax information to evolve the position of the source assuming straight-line motion:

```
>>> c_2mass_epoch = c.apply_space_motion(epoch_2mass)
```

Now that we have the coordinates of the TGAS source at the 2MASS epoch, we can do the cross-match (see also Separations, Offsets, Catalog Matching, and Related Functionality):

```
>>> idx, sep, _ = c_2mass_epoch.match_to_catalog_sky(catalog_2mass)
>>> sep[0].to_string()
'0d00m00.2818s'
>>> idx
array(5)
```

The closest source it found is 0.2818 arcseconds away and corresponds to row index 5 in the 2MASS catalog. We can then, for example, compute *Gaia*-2MASS colors:

```
>>> G = result_tgas['phot_g_mean_mag']
>>> J = result_2mass['Jmag'][idx]
>>> K = result_2mass['Kmag'][idx]
>>> G - J, G - K
(1.0151743233481962, 1.3731746233481958)
```

Using the SpectralCoord Class

Warning

The `SpectralCoord` class is new in Astropy v4.1 and should be considered experimental at this time. Note that we do not fully support cases where the observer and target are moving relativistically relative to each other, so care should be taken in those cases. It is possible that there will be API changes in future versions of Astropy based on user feedback. If you have specific ideas for how it might be improved, please let us know on the astropy-dev mailing list or at http://feedback.astropy.org.

The `SpectralCoord` class provides an interface for representing and transforming spectral coordinates such as frequencies, wavelengths, and photon energies, as well as equivalent Doppler velocities. While the plain `Quantity` class can also represent these kinds of physical quantities, and allow conversion via dedicated equivalencies (such as u.spectral or the u.doppler_* equivalencies), **SpectralCoord** (which is a sub-class of **Quantity**) aims to make this more straightforward, and can also be made aware of the observer and target reference frames, allowing for example transformation from telescope-centric (or topocentric) frames to e.g. Barycentric or Local Standard of Rest (LSRK and LSRD) velocity frames.

Creating SpectralCoord Objects

Since the **SpectralCoord** class is a sub-class of **Quantity**, the simplest way to initialize it is to provide a value (or values) and a unit, or an existing **Quantity**:

```
>>> from astropy import units as u
>>> from astropy.coordinates import SpectralCoord
>>> sc1 = SpectralCoord(34.2, unit='GHz')
>>> sc1
<SpectralCoord 34.2 GHz>
>>> sc2 = SpectralCoord([654.2, 654.4, 654.6] * u.nm)
>>> sc2
<SpectralCoord [654.2, 654.4, 654.6] nm>
```

At this point, we are not making any assumptions about the observer frame, or the target that is being observed. As we will see in subsequent sections, more information can be provided when initializing **SpectralCoord** objects, but first we take a look at simple unit conversions with these objects.

Unit conversion

By default, unit conversions between spectral units will work without having to specify the u.spectral equivalency:

```
>>> sc2.to(u.micron)
<SpectralCoord [0.6542, 0.6544, 0.6546] micron>
>>> sc2.to(u.eV)
<SpectralCoord [1.89520328, 1.89462406, 1.89404519] eV>
>>> sc2.to(u.THz)
<SpectralCoord [458.25811373, 458.11805929, 457.97809044] THz>
```

As is the case with **Quantity** and the Doppler equivalencies, it is also posible to convert these absolute spectral coordinates into velocities, assuming a particular rest frequency or wavelength (such as that of a spectral line). For example, to convert the above values into velocities relative to the Halpha line at 656.65 nm, assuming the optical Doppler convention, you can do:

```
>>> sc3 = sc2.to(u.km / u.s,
...              doppler_convention='optical',
...              doppler_rest=656.65 * u.nm)
>>> sc3
<SpectralCoord
   (doppler_rest=656.65 nm
    doppler_convention=optical)
  [-1118.5433977 , -1027.23373258,  -935.92406746] km / s>
```

The rest value for the Doppler conversion as well as the convention to use are stored in the resulting `sc3` **`SpectralCoord`** object. You can then convert back to frequency without having to specify them again:

```
>>> sc3.to(u.THz)
<SpectralCoord
   (doppler_rest=656.65 nm
    doppler_convention=optical)
  [458.25811373, 458.11805929, 457.97809044] THz>
```

or you can explicitly specify a different convention or rest value to use:

```
>>> sc3.to(u.km / u.s, doppler_convention='relativistic')
<SpectralCoord
   (doppler_rest=656.65 nm
    doppler_convention=relativistic)
  [-1120.63005892, -1028.99362163,  -937.38499411] km / s>
```

It is also possible to set `doppler_convention` and `doppler_rest` from the start, even when creating a **`SpectralCoord`** in frequency, energy, or wavelength:

```
>>> sc4 = SpectralCoord(343 * u.GHz,
...                     doppler_convention='radio',
...                     doppler_rest=342.91 * u.GHz)
>>> sc4.to(u.km / u.s)
<SpectralCoord
   (doppler_rest=342.91 GHz
    doppler_convention=radio)
  -78.68338987 km / s>
```

Reference frame transformations

If you work with any kind of spectral data, you will often need to determine and/or apply velocity corrections due to different frames of reference, or apply or remove the effects of redshift. There are two main ways to do this using the `SpectralCoord` class:

- You can specify or change the velocity offset or redshift between the observer and the target without having to specify the absolute observer and target, but rather specify a velocity difference. For example, that you know that there is a velocity difference of 15km/s along the line of sight, or that you are observing a galaxy at z=3.2. This can be useful for quick analysis but will not determine any frame transformations (e.g. from topocentric to barycentric) for you.
- You can specify the absolute position of the observer and the target, as well as the date of observation, which means that `SpectralCoord` can then compute different frame transformations. If information about the observer and target are available, this is the recommended approach, although it requires you to specify more information when setting up the `SpectralCoord`

In the next two sections we will look at each of these in turn.

Specifying radial velocity or redshift manually

As an example, we will consider an example of a `SpectralCoord` which represents frequencies which form the x-axis of a (small) spectrum. We happen to know that the target that was observed appears to be at a redshift of z=0.5, and we will assume that any frequency shifts due to the Earth's motion are unimportant. In the reference frame of the telescope, the spectrometer provides 10 values between 500 and 900nm:

```
>>> import numpy as np
>>> wavs = SpectralCoord(np.linspace(500, 900, 9) * u.nm,
redshift=0.5)
>>> wavs
<SpectralCoord
   (observer to target:
      radial_velocity=115304.79153846153 km / s
      redshift=0.5)
  [500., 550., 600., 650., 700., 750., 800., 850., 900.] nm>
```

We have set redshift=0.5 here so that we can keep track of what frame of reference our spectral values are in. The `radial_velocity` property gives the recession velocity equivalent to that redshift, and it is indeed large enough that we don't need to worry about the rotation of the Earth on itself around the

Sun (which would be at most a ~30km/s contribution).

Note

In the context of **SpectralCoord**, we use the full relativistic relation between redshift and velocity, i.e. \(1 + z = \sqrt{(1 + v/c)/(1 - v/c)}\)

We now want to shift the wavelengths so that they would be in the rest frame of the galaxy. We can do this using the **to_rest()** method:

```
>>> wavs_rest = wavs.to_rest()
>>> wavs_rest
<SpectralCoord
   (observer to target:
      radial_velocity=0.0 km / s
      redshift=0.0)
  [333.33333333, 366.66666667, 400.        , 433.33333333,
466.66666667,
   500.        , 533.33333333, 566.66666667, 600.        ] nm>
```

The wavelengths have decreased by 1/3, which is what we expect for z=0.5. Note that the `redshift` and `radial_velocity` properties are now zero, since we are in the reference frame of the target. We can also use the **with_radial_velocity_shift()** method to more generically apply redshift and velocity corrections. The simplest way to use this method is to give a single value that will be applied to the target - if this value does not have units, it is interpreted as a redshift:

```
>>> wavs_orig = wavs_rest.with_radial_velocity_shift(0.5)
>>> wavs_orig
<SpectralCoord
   (observer to target:
      radial_velocity=115304.79153846153 km / s
      redshift=0.5)
  [500., 550., 600., 650., 700., 750., 800., 850., 900.] nm>
```

This returns an object equivalent to the one we started with, since we've re-applied a redshift of 0.5. We could also provide a velocity as a **Quantity**:

```
>>> wavs_rest.with_radial_velocity_shift(100000 * u.km / u.s)
<SpectralCoord
   (observer to target:
      radial_velocity=100000.0 km / s
      redshift=0.41458078170200463)
  [471.52692723, 518.67961996, 565.83231268, 612.9850054 ,
660.13769813,
   707.29039085, 754.44308357, 801.5957763 , 848.74846902] nm>
```

which shifts the values to a frame of reference at a redshift of approximately 0.33 (that is, if the spectrum did contain a contribution from an object at z=0.33, these would be the rest wavelengths for that object.

Specifying an observer and a target explicitly

To use the more advanced functionality in **SpectralCoord**, including the ability to easily transform between different well-defined velocity frames, you will need to give it information about the location (and optionally velocity) of the observer and target. This is done by passing either coordinate frame objects or **SkyCoord** objects. To take a concrete example, let's assume that we are now observe the source T Tau using the ALMA telescope. To create an observer object corresponding to this, we can make use of the **EarthLocation** class:

```
>>> from astropy.coordinates import EarthLocation
>>> location = EarthLocation.of_site('ALMA')
>>> location
<EarthLocation (2225015.30883296, -5440016.41799762,
-2481631.27428014) m>
```

The three values in meters are geocentric coordinates, i.e. the 3D coordinates relative to the center of the Earth. See **EarthLocation** for more details about the different ways of creating these kinds of objects.

Once you have done this, you will need to convert `location` to a coordinate object using the **get_itrs()** method, which takes the observation time (which is important to know for any kind of velocity frame transformation):

```
>>> from astropy.time import Time
>>> alma = location.get_itrs(obstime=Time('2019-04-24T02:32:10'))
>>> alma
<ITRS Coordinate (obstime=2019-04-24T02:32:10.000): (x, y, z) in m
    (2225015.30883296, -5440016.41799762, -2481631.27428014)>
```

ITRS here stands for International Terrestrial Reference System which is a 3D coordinate frame centered on the Earth's center and rotating with the Earth, so the observatory will be stationary in this frame of reference.

For the target, the simplest way is to use the **SkyCoord** class:

```
>>> from astropy.coordinates import SkyCoord
>>> ttau = SkyCoord('04h21m59.43s +19d32m06.4', frame='icrs',
...                 radial_velocity=23.9 * u.km / u.s,
...                 distance=144.321 * u.pc)
```

In this case we specified a radial velocity and a distance for the target (using the T Tauri SIMBAD entry, but it is also possible to not specify these, which means the target is assumed to be stationary in the frame in which it is observed, and are assumed to be at large distance from the Sun (such that any parallax effects would be unimportant if relevant). The radial velocity is assumed to be in the frame used to define the target location, so it is relative to the ICRS origin (the Solar System barycenter) in the above case.

We now define a set of frequencies corresponding to the channels in which fluxes have been measured (for the purposes of the example here we will assume we have only 11 frequencies):

```
>>> sc_ttau = SpectralCoord(np.linspace(200, 300, 11) * u.GHz,
...                         observer=alma, target=ttau)
>>> sc_ttau
<SpectralCoord
   (observer: <ITRS Coordinate (obstime=2019-04-24T02:32:10.000): (x,
y, z) in m
                  (2225015.30883296, -5440016.41799762,
-2481631.27428014)
               (v_x, v_y, v_z) in km / s
                  (0., 0., 0.)>
    target: <ICRS Coordinate: (ra, dec, distance) in (deg, deg, pc)
                (65.497625, 19.53511111, 144.321)
             (pm_ra_cosdec, pm_dec, radial_velocity) in (mas / yr,
mas / yr, km / s)
                (1.37949782e-15, 1.46375638e-15, 23.9)>
    observer to target (computed from above):
      radial_velocity=41.03594947739035 km / s
      redshift=0.00013689056309340586)
  [200., 210., 220., 230., 240., 250., 260., 270., 280., 290., 300.]
GHz>
```

We can already see above that **SpectralCoord** has computed the difference in velocity between the observatory and T Tau, which includes the motion of the observatory around the Earth, the motion of the Earth around the Solar System barycenter, and the radial velocity of T Tau relative to the Solar System barycenter. We can get this value directly with:

```
>>> sc_ttau.radial_velocity
<Quantity 41.03594948 km / s>
```

If you work with any kind of spectral data, you will often need to determine and/or apply velocity corrections due to different frames of reference. For example if you have observations of the same object on the sky taken at different dates, it is common to transform these to a common velocity frame of reference, so that your spectral coordinates are those that would have applied if the observer had been stationary relative to e.g. the Solar System Barycenter. You may also want to transform your spectral coordinates so that they would be in a frame at rest relative to the local standard of rest (LSR), the center of the Milky Way, the Local Group, or even the Cosmic Microwave Background (CMB) dipole.

We can transform our frequencies for the observations of T Tau to different velocity frames using the **`with_observer_stationary_relative_to()`** method. This method can take the name of an existing coordinate/velocity frame, a **`BaseCoordinateFrame`** instance, or any arbitrary 3D position and velocity coordinate object defined either as a **`BaseCoordinateFrame`** or a **`SkyCoord`** object. Most commonly-used frames are accessible using strings. For example to transform to a velocity frame stationary with respect to the center of the Earth (so removing the effect of the Earth's rotation), we can use the `'gcrs'` which stands for *Geocentric Celestial Reference System* (GCRS):

```
>>> sc_ttau.with_observer_stationary_relative_to('gcrs')
<SpectralCoord
   (observer: <GCRS Coordinate (obstime=2019-04-24T02:32:10.000,
obsgeoloc=(0., 0., 0.) m, obsgeovel=(0., 0., 0.) m / s): (x, y, z) in
m
                  (-5878853.86171412, -192921.84773269,
-2470794.19765021)
               (v_x, v_y, v_z) in km / s
                  (4.33251262e-09, 8.96175625e-08, -1.49258412e-08)>
    target: <ICRS Coordinate: (ra, dec, distance) in (deg, deg, pc)
                (65.497625, 19.53511111, 144.321)
             (pm_ra_cosdec, pm_dec, radial_velocity) in (mas / yr,
mas / yr, km / s)
                (1.37949782e-15, 1.46375638e-15, 23.9)>
    observer to target (computed from above):
      radial_velocity=40.674086368345165 km / s
      redshift=0.00013568335316072044)
  [200.00024141, 210.00025348, 220.00026555, 230.00027762,
240.00028969,
   250.00030176, 260.00031383, 270.0003259 , 280.00033797,
290.00035004,
   300.00036211] GHz>
```

As you can see, the frequencies have changed slightly, which is because we have removed the Doppler shift caused by the Earth's rotation (this can also be seen in the `radial_velocity` property, which has changed by ~0.35 km/s. To use a velocity reference frame relative to the Solar System barycenter, which is the origin of the *International Celestial Reference System* (ICRS) system, we can use:

```
>>> sc_ttau.with_observer_stationary_relative_to('icrs')
<SpectralCoord
   (observer: <ICRS Coordinate: (x, y, z) in m
                  (-1.25867767e+11, -7.48979688e+10, -3.24757657e+10)
               (v_x, v_y, v_z) in km / s
                  (0., 0., 0.)>
    target: <ICRS Coordinate: (ra, dec, distance) in (deg, deg, pc)
                (65.497625, 19.53511111, 144.321)
             (pm_ra_cosdec, pm_dec, radial_velocity) in (mas / yr,
mas / yr, km / s)
                (1.37949782e-15, 1.46375638e-15, 23.9)>
    observer to target (computed from above):
      radial_velocity=23.9 km / s
      redshift=7.97249967898761e-05)
  [200.0114322 , 210.01200381, 220.01257542, 230.01314703,
240.01371864,
   250.01429025, 260.01486186, 270.01543347, 280.01600508,
290.01657669,
   300.0171483 ] GHz>
```

Note that in this case the total radial velocity between the observer and the target matches what we specified when we set up the target, since it was defined relative to the ICRS origin (the Solar System barycenter). The observer location is still as before, but the observer velocity is now ~10-20 km/s in x, y, and z, which is because the observer is now stationary relative to the barycenter so has a significant velocity relative to the surface of the Earth.

We can also transform the frequencies to the Kinematic Local Standard of Rest (LSRK) frame of reference, which is a reference frame commonly used in some branches of astronomy (such as radio astronomy):

```
>>> sc_ttau.with_observer_stationary_relative_to('lsrk')
<SpectralCoord
   (observer: <LSRK Coordinate: (x, y, z) in m
                  (-1.25867767e+11, -7.48979688e+10, -3.24757657e+10)
               (v_x, v_y, v_z) in km / s
                  (0., 0., 0.)>
    target: <ICRS Coordinate: (ra, dec, distance) in (deg, deg, pc)
                (65.497625, 19.53511111, 144.321)
             (pm_ra_cosdec, pm_dec, radial_velocity) in (mas / yr,
mas / yr, km / s)
                (1.37949782e-15, 1.46375638e-15, 23.9)>
    observer to target (computed from above):
      radial_velocity=12.50698856018455 km / s
      redshift=4.171969349386906e-05)
  [200.01903338, 210.01998505, 220.02093672, 230.02188839,
240.02284006,
   250.02379172, 260.02474339, 270.02569506, 280.02664673,
290.0275984 ,
   300.02855007] GHz>
```

See Common velocity frames for a list of common velocity frames available as strings on the **`SpectralCoord`** class.

Since we can give any arbitrary **`SkyCoord`** to the **`with_observer_stationary_relative_to()`** method, we can also specify the target itself, to find the frequencies in the rest frame of the target:

```
>>> sc_ttau_targetframe =
sc_ttau.with_observer_stationary_relative_to(sc_ttau.target)
>>> sc_ttau_targetframe
<SpectralCoord
   (observer: <ICRS Coordinate: (x, y, z) in m
                  (-1.25867767e+11, -7.48979688e+10, -3.24757657e+10)
```

```
                (v_x, v_y, v_z) in km / s
                   (9.34149908, 20.49579745, 7.99178839)>
    target: <ICRS Coordinate: (ra, dec, distance) in (deg, deg, pc)
                (65.497625, 19.53511111, 144.321)
             (pm_ra_cosdec, pm_dec, radial_velocity) in (mas / yr,
mas / yr, km / s)
                (1.37949782e-15, 1.46375638e-15, 23.9)>
    observer to target (computed from above):
      radial_velocity=0.0 km / s
      redshift=0.0)
  [200.02737811, 210.02874702, 220.03011592, 230.03148483,
240.03285374,
   250.03422264, 260.03559155, 270.03696045, 280.03832936,
290.03969826,
   300.04106717] GHz>
```

The `radial_velocity`, which is the velocity offset between observer and target, is now zero.

`SpectralCoord` is intended to be versatile and be useful for representing any spectral values - not just the x-axis of a spectrum, but also for example the frequencies of spectral features. For example, if we now consider that we found a spectral feature that appears to have components at the following frequencies in the frame of reference of the telescope:

```
>>> sc_feat = SpectralCoord([115.26, 115.266, 115.267] * u.GHz,
...                          observer=alma, target=ttau)
```

We can convert these to the rest frame of the target using:

```
>>> sc_feat_rest =
sc_feat.with_observer_stationary_relative_to(sc_feat.target)
>>> sc_feat_rest
<SpectralCoord
   (observer: <ICRS Coordinate: (x, y, z) in m
                   (-1.25867767e+11, -7.48979688e+10, -3.24757657e+10)
                (v_x, v_y, v_z) in km / s
                   (9.34149908, 20.49579745, 7.99178839)>
    target: <ICRS Coordinate: (ra, dec, distance) in (deg, deg, pc)
                (65.497625, 19.53511111, 144.321)
             (pm_ra_cosdec, pm_dec, radial_velocity) in (mas / yr,
mas / yr, km / s)
                (1.37949782e-15, 1.46375638e-15, 23.9)>
    observer to target (computed from above):
      radial_velocity=0.0 km / s
      redshift=0.0)
  [115.27577801, 115.28177883, 115.28277896] GHz>
```

The frequencies are very close to the rest frequency of the 12CO J=1-0 molecular line transition, which is 115.2712018 GHz. However, they are not exactly the same, so if the features we see are indeed from 12CO, then they are Doppler shifted compared to what we consider the rest frame of T Tau. We can convert these frequencies to velocities assuming the Doppler shift equation (in this case with the radio convention):

```
>>> sc_feat_rest.to(u.km / u.s, doppler_convention='radio',
doppler_rest=115.27120180 * u.GHz)
<SpectralCoord
   (observer: <ICRS Coordinate: (x, y, z) in m
                  (-1.25867767e+11, -7.48979688e+10, -3.24757657e+10)
               (v_x, v_y, v_z) in km / s
                  (9.34149908, 20.49579745, 7.99178839)>
    target: <ICRS Coordinate: (ra, dec, distance) in (deg, deg, pc)
                (65.497625, 19.53511111, 144.321)
             (pm_ra_cosdec, pm_dec, radial_velocity) in (mas / yr,
mas / yr, km / s)
                (1.37949782e-15, 1.46375638e-15, 23.9)>
    observer to target (computed from above):
      radial_velocity=0.0 km / s
      redshift=0.0
    doppler_rest=115.2712018 GHz
    doppler_convention=radio)
  [-11.90160347, -27.50828539, -30.10939904] km / s>
```

Note that these resulting velocities are different from the `radial_velocity` property (which is still zero here) - the latter is the difference in velocity between observer and target, while the former are how much the spectral values are Doppler shifted by relative to the rest frequency or wavelength.

So if the features are indeed from 12CO, they have velocities of approximately -11.9, -27.5 and -30.1 km/s relative to the T tau rest frame.

Common velocity frames

Any valid astropy coordinate frame can be passed to the **`with_observer_stationary_relative_to()`** method, including string aliases such as `icrs`. Below we list some of the frames commonly used to define spectral coordinates in:

The velocity frames available as constants on the **`SpectralCoord`** class are:

Frame name	Description

Frame name	Description
`'gcrs'`	Geocentric frame (defined as stationary relative to the GCRS origin)
`'icrs'`	Barycentric frame (defined as stationary relative to the ICRS origin)
`'hcrs'`	Heliocentric frame (defined as stationary relative to the HCRS origin)
`'lsrk'`	Kinematic Local Standard of Rest (LSRK), defined as having a velocity of 20 km/s towards 18h +30d (B1900) relative to the Solar System Barycenter [1].
`'lsrd'`	Dynamical Local Standard of Rest (LSRD), defined as having a velocity of U=9 km/s, V=12 km/s, and W=7 km/s in Galactic coordinates (equivalent to 16.552945 km/s towards l=53.13 and b=25.02) [2].
`'lsr'`	A more recent definition of the Local Standard of rest, with U=11.1 km/s, V=12.24 km/s, and W=7.25 km/s in Galactic coordinates [3].

Defining custom velocity frames

As mentioned in the earlier examples on this page, it is possible to pass any arbitrary **`BaseCoordinateFrame`** or **`SkyCoord`** object to the **`with_observer_stationary_relative_to()`** method, and the observer will be updated to be stationary relative to those coordinates. As an example, we can define an object that can be used to define a velocity frame that moves with the local group of galaxies. There is not a unique definition of this, but for the purposes of this example we use the IAU 1976-recommended value which states that the Solar System barycenter is moving at 300 km/s towards l=90 and b=0 in the velocity frame of the local group of galaxies [4]. Given this value, we can define the velocity frame using:

```
>>> from astropy.coordinates import Galactic
>>> localgroup_frame = Galactic(u=0 * u.km, v=0 * u.km, w=0 * u.km,
...                             U=0 * u.km / u.s, V=-300 * u.km /
u.s, W=0 * u.km / u.s,
...                             representation_type='cartesian',
...                             differential_type='cartesian')
```

Note that here we specify the velocity as -300, because what we need here is the velocity of the local group relative to the Solar System barycenter. With this object, we can then transform a **`SpectralCoord`** so that the observer is stationary in that frame of reference:

```
>>> sc_ttau.with_observer_stationary_relative_to(localgroup_frame)
<SpectralCoord
```

```
   (observer: <Galactic Coordinate: (u, v, w) in m
                  (8.8038652e+10, -5.31344273e+10, 1.09238291e+11)
               (U, V, W) in km / s
                  (-1.42108547e-14, -300., 2.84217094e-14)>
    target: <ICRS Coordinate: (ra, dec, distance) in (deg, deg, pc)
                (65.497625, 19.53511111, 144.321)
             (pm_ra_cosdec, pm_dec, radial_velocity) in (mas / yr,
mas / yr, km / s)
                (1.37949782e-15, 1.46375638e-15, 23.9)>
    observer to target (computed from above):
      radial_velocity=42.33062895275233 km / s
      redshift=0.00014120974955456056)
  [199.99913628, 209.9990931 , 219.99904991, 229.99900673,
239.99896354,
   249.99892036, 259.99887717, 269.99883398, 279.9987908 ,
289.99874761,
   299.99870443] GHz>
```

References

[1] Meeks, M. L. 1976, *Methods of experimental physics. Vol._12. Astrophysics. Part C: Radio observations*, Section 6.1 by Gordon, M. A. [ADS].

[2] Delhaye, J. 1965, *Galactic Structure*. Edited by Adriaan Blaauw and Maarten Schmidt. Published by the University of Chicago Press, p61 [ADS].

[3] Schönrich, R., Binney, J., & Dehnen, W. 2010, MNRAS, 403, 1829 [ADS].

[4] *Transactions of the IAU Vol. XVI B Proceedings of the 16th General Assembly, Reports of Meetings of Commissions: Comptes Rendus Des Séances Des Commissions, Commission 28*. [DOI]

Description of the Galactocentric Coordinate Frame

While many other frames implemented in **`astropy.coordinates`** are standardized in some way (e.g., defined by the IAU), there is no standard Milky Way reference frame with the center of the Milky Way as its origin. (This is distinct from **`Galactic`** coordinates, which point toward the Galactic Center but have their origin in the Solar System). The **`Galactocentric`** frame class is meant to be flexible enough to support all common definitions of such a transformation, but with reasonable default parameter values, such as the solar velocity relative to the Galactic center, the solar height above the Galactic midplane, etc. Below, we describe our generalized definition of the transformation from the ICRS to/from Galactocentric coordinates, and describe how to customize the default Galactocentric parameters that are used when the

`Galactocentric` frame is initialized without explicitly passing in parameter values.

Definition of the Transformation

This document describes the mathematics behind the transformation from `ICRS` to `Galactocentric` coordinates. This is described in detail here on account of the mathematical subtleties and the fact that there is no official standard/definition for this frame. For examples of how to use this transformation in code, see the the *Examples* section of the `Galactocentric` class documentation.

We assume that we start with a 3D position in the ICRS reference frame: a Right Ascension, Declination, and heliocentric distance, \((\alpha, \delta, d)\). We can convert this to a Cartesian position using the standard transformation from Cartesian to spherical coordinates:

\[\begin{split}\begin{aligned} x_{\rm icrs} &= d\cos{\alpha}\cos{\delta}\\ y_{\rm icrs} &= d\sin{\alpha}\cos{\delta}\\ z_{\rm icrs} &= d\sin{\delta}\\ \boldsymbol{r}_{\rm icrs} &= \begin{pmatrix} x_{\rm icrs}\\ y_{\rm icrs}\\ z_{\rm icrs} \end{pmatrix}\end{aligned}\end{split}\]
The first transformation rotates the \(x_{\rm icrs}\) axis so that the new \(x'\) axis points towards the Galactic Center (GC), specified by the ICRS position \((\alpha_{\rm GC}, \delta_{\rm GC})\) (in the `Galactocentric` frame, this is controlled by the frame attribute `galcen_coord`):

\[\begin{split}\begin{aligned} \boldsymbol{R}_1 &= \begin{bmatrix} \cos\delta_{\rm GC}& 0 & \sin\delta_{\rm GC}\\ 0 & 1 & 0 \\ -\sin\delta_{\rm GC}& 0 & \cos\delta_{\rm GC}\end{bmatrix}\\ \boldsymbol{R}_2 &= \begin{bmatrix} \cos\alpha_{\rm GC}& \sin\alpha_{\rm GC}& 0\\ -\sin\alpha_{\rm GC}& \cos\alpha_{\rm GC}& 0\\ 0 & 0 & 1 \end{bmatrix}.\end{aligned} \end{split}\]
The transformation thus far has aligned the \(x'\) axis with the vector pointing from the Sun to the GC, but the \(y'\) and \(z'\) axes point in arbitrary directions. We adopt the orientation of the Galactic plane as the normal to the north pole of Galactic coordinates defined by the IAU (Blaauw et. al. 1960). This extra "roll" angle, \(\eta\), was measured by transforming a grid of points along \(l=0\) to this interim frame and minimizing the square of their \(y'\) positions. We find:

\[\begin{split}\begin{aligned} \eta &= 58.5986320306^\circ\\ \boldsymbol{R}_3 &= \begin{bmatrix} 1 & 0 & 0\\ 0 & \cos\eta & \sin\eta\\ 0 & -\sin\eta & \cos\eta \end{bmatrix} \end{aligned}\end{split}\]
The full rotation matrix thus far is:

\[\begin{split}\begin{gathered} \boldsymbol{R} = \boldsymbol{R}_3 \boldsymbol{R}_1 \boldsymbol{R}_2 = \\ \begin{bmatrix} \cos\alpha_{\rm GC}\cos\delta_{\rm GC}&

\cos\delta_{\rm GC}\sin\alpha_{\rm GC}& -\sin\delta_{\rm GC}\\ \cos\alpha_{\rm GC}\sin \delta_{\rm GC}\sin\eta - \sin\alpha_{\rm GC}\cos\eta & \sin\alpha_{\rm GC}\sin\delta_{\rm GC}\sin\eta + \cos\alpha_{\rm GC}\cos\eta & \cos\delta_{\rm GC}\sin\eta\\ \cos\alpha_{\rm GC}\sin\delta_{\rm GC}\cos\eta + \sin\alpha_{\rm GC}\sin\eta & \sin\alpha_{\rm GC}\sin \delta_{\rm GC}\cos\eta - \cos\alpha_{\rm GC}\sin\eta & \cos\delta_{\rm GC}\cos\eta \end{bmatrix}\end{gathered}\end{split}\]

With the rotated position vector \(\boldsymbol{R}\boldsymbol{r}_{\rm icrs}\), we can now subtract the distance to the GC, \(d_{\rm GC}\), which is purely along the \(x'\) axis:

\[\begin{aligned} \boldsymbol{r}' &= \boldsymbol{R}\boldsymbol{r}_{\rm icrs} - d_{\rm GC}\hat{\boldsymbol{x}}_{\rm GC}.\end{aligned}\]

where \(\hat{\boldsymbol{x}}_{\rm GC} = (1,0,0)^{\mathsf{T}}\).

The final transformation accounts for the (specified) height of the Sun above the Galactic midplane by rotating about the final \(y''\) axis by the angle \(\theta= \sin^{-1}(z_\odot / d_{\rm GC})\):

\[\begin{split}\begin{aligned} \boldsymbol{H} &= \begin{bmatrix} \cos\theta & 0 & \sin\theta\\ 0 & 1 & 0\\ -\sin\theta & 0 & \cos\theta \end{bmatrix}\end{aligned}\end{split}\]

where \(z_\odot\) is the measured height of the Sun above the midplane.

The full transformation is then:

\[\boldsymbol{r}_{\rm GC} = \boldsymbol{H} \left(\boldsymbol{R}\boldsymbol{r}_{\rm icrs} - d_{\rm GC}\hat{\boldsymbol{x}}_{\rm GC}\right).\]

Examples:

For an example of how to use the `Galactocentric` frame, see Transforming positions and velocities to and from a Galactocentric frame.

Controlling the Default Frame Parameters

All of the frame-defining parameters of the `Galactocentric` frame are customizable and can be set by passing arguments in to the `Galactocentric` initializer. However, it is often convenient to use the frame without having to pass in every parameter. Hence, the class comes with reasonable default values for these parameters, but more precise measurements of the solar position or motion in the Galaxy are constantly being made. The default values of the `Galactocentric` frame attributes will therefore be updated as necessary with subsequent releases of `astropy`. We therefore provide a mechanism to globally or locally control the default parameter values used in this frame through the `galactocentric_frame_defaults` `ScienceState` class.

The `galactocentric_frame_defaults` class controls the default parameter settings in `Galactocentric` by mapping a set of string names to particular choices of the parameter values. For an up-to-date list of valid names, see the docstring of `galactocentric_frame_defaults`, but these names are things like `'pre-v4.0'`, which sets the default parameter values to their original definition (i.e. pre-astropy-v4.0) values, and `'v4.0'`, which sets the default parameter values to a more modern set of measurements as updated in Astropy version 4.0. Also, custom sets of measurements can be registered to `galactocentric_frame_defaults` and used like the built-in options.

`galactocentric_frame_defaults` also tracks the references (i.e. scientific papers that define the parameter values) for all parameter values, as well as any further specified metadata information.

As with other `ScienceState` subclasses, the `galactocentric_frame_defaults` class can be used to globally set the frame defaults at runtime.

Examples

The default parameter values can be seen by initializing the `Galactocentric` frame with no arguments:

```
>>> from astropy.coordinates import Galactocentric
>>> Galactocentric()
<Galactocentric Frame (galcen_coord=<ICRS Coordinate: (ra, dec) in
deg
    (266.4051, -28.936175)>, galcen_distance=8.122 kpc, galcen_v_sun=
(12.9, 245.6, 7.78) km / s, z_sun=20.8 pc, roll=0.0 deg)>
```

These default values can be modified using this class:

```
>>> from astropy.coordinates import galactocentric_frame_defaults
>>> _ = galactocentric_frame_defaults.set('v4.0')
>>> Galactocentric()
<Galactocentric Frame (galcen_coord=<ICRS Coordinate: (ra, dec) in
deg
    (266.4051, -28.936175)>, galcen_distance=8.122 kpc, galcen_v_sun=
(12.9, 245.6, 7.78) km / s, z_sun=20.8 pc, roll=0.0 deg)>
>>> _ = galactocentric_frame_defaults.set('pre-v4.0')
>>> Galactocentric()
<Galactocentric Frame (galcen_coord=<ICRS Coordinate: (ra, dec) in
deg
    (266.4051, -28.936175)>, galcen_distance=8.3 kpc, galcen_v_sun=
(11.1, 232.24, 7.25) km / s, z_sun=27.0 pc, roll=0.0 deg)>
```

The default parameters can also be updated by using this class as a context manager to change the default parameter values locally to a piece of your code:

```
>>> with galactocentric_frame_defaults.set('pre-v4.0'):
...     print(Galactocentric())
<Galactocentric Frame (galcen_coord=<ICRS Coordinate: (ra, dec) in
deg
    (266.4051, -28.936175)>, galcen_distance=8.3 kpc, galcen_v_sun=
(11.1, 232.24, 7.25) km / s, z_sun=27.0 pc, roll=0.0 deg)>
```

Again, changing the default parameter values will not affect frame attributes that are explicitly specified:

```
>>> import astropy.units as u
>>> with galactocentric_frame_defaults.set('pre-v4.0'):
...     print(Galactocentric(galcen_distance=8.0*u.kpc))
<Galactocentric Frame (galcen_coord=<ICRS Coordinate: (ra, dec) in
deg
    (266.4051, -28.936175)>, galcen_distance=8.0 kpc, galcen_v_sun=
(11.1, 232.24, 7.25) km / s, z_sun=27.0 pc, roll=0.0 deg)>
```

Additional parameter sets may be registered, for instance to use the Dehnen & Binney (1998) measurements of the solar motion. We can also add metadata, such as the 1-sigma errors:

```
>>> state = galactocentric_frame_defaults.get_from_registry("v4.0")
>>> state["parameters"]["galcen_v_sun"] = (10.00, 225.25, 7.17) *
(u.km / u.s)
>>> state["references"]["galcen_v_sun"] =
"http://www.adsabs.harvard.edu/full/1998MNRAS.298..387D"
>>> state["error"] = {"galcen_v_sun": (0.36, 0.62, 0.38) * (u.km /
u.s)}
>>> galactocentric_frame_defaults.register(name="DB1998", **state)
```

Just as in the previous examples, the new parameter set can be get / set:

```
>>> state = galactocentric_frame_defaults.get_from_registry("DB1998")
>>> print(state["error"]["galcen_v_sun"])
[0.36 0.62 0.38] km / s
```

Starting with Astropy v4.1, unless set with the `galactocentric_frame_defaults` class, the default parameter values for the `Galactocentric` frame are now set to `'latest'`, meaning that the default parameter values may change if you update Astropy. If you use the `Galactocentric` frame without specifying all parameter values explicitly, we therefore suggest manually setting the frame default set manually in any science code that depends sensitively on the choice of, e.g., solar motion or the other frame parameters. For example, in such code, we recommend adding something like this to your import block (here using `'v4.0'` as an example):

```
>>> import astropy.coordinates as coord
>>> coord.galactocentric_frame_defaults.set('v4.0')
```

Usage Tips/Suggestions for Methods That Access Remote Resources

There are currently two methods that rely on getting remote data to work.

The first is the `SkyCoord from_name()` method, which uses Sesame to retrieve coordinates for a particular named object:

```
>>> from astropy.coordinates import SkyCoord
>>> SkyCoord.from_name("PSR J1012+5307")
<SkyCoord (ICRS): (ra, dec) in deg
    ( 153.1393271,  53.117343)>
```

The second is the `EarthLocation of_site()` method, which provides a similar quick way to get an `EarthLocation` from an observatory name:

```
>>> from astropy.coordinates import EarthLocation
>>> apo = EarthLocation.of_site('Apache Point Observatory')
>>> apo
<EarthLocation (-1463969.3018517173, -5166673.342234327,
3434985.7120456537) m>
```

The full list of available observatory names can be obtained with `astropy.coordinates.EarthLocation.get_site_names()`.

While these methods are convenient, there are several considerations to take into account:

- Since these methods access online data, the data may evolve over time (for example, the accuracy of coordinates might improve, and new observatories may be added). Therefore, this means that a script using these and currently running may give a different answer in five years. Therefore, users concerned with reproducibility should not use these methods in their final scripts, but can instead use them to get the values required, and then hard-code them into the scripts. For example, we can check the coordinates of the Kitt Peak Observatories using:

```
>>> loc = EarthLocation.of_site('Kitt Peak')
```

Note that this command requires an internet connection.

We can then view the actual Cartesian coordinates for the observatory:

```
>>> loc
<EarthLocation (-1994502.6043061386, -5037538.54232911,
3358104.9969029757) m>
```

This can then be converted into code:

```
>>> loc = EarthLocation(-1994502.6043061386, -5037538.54232911,
3358104.9969029757, unit='m')
```

This latter line can then be included in a script and will ensure that the results stay the same over time.

- The online data may not be accurate enough for your purposes. If maximum accuracy is paramount, we recommend that you determine the celestial or Earth coordinates yourself and hard-code these, rather than using the convenience methods.
- These methods will not function if an internet connection is not available. Therefore, if you need to work on a script while offline, follow the instructions in the first bullet point above to hard-code the coordinates before going offline.

Important Definitions

For reference, below, we define some key terms as they are used in **`coordinates`**, due to some ambiguities that exist in the colloquial use of these terms. Chief among these terms is the concept of a "coordinate system." To some members of the community, "coordinate system" means the *representation* of a point in space (e.g., "Cartesian coordinate system" is different from "Spherical polar coordinate system"). Another use of "coordinate system" is to mean a unique reference frame with a particular set of reference points (e.g., "the ICRS coordinate system" or the "J2000 coordinate system"). This second meaning is further complicated by the fact that such systems use quite different ways of defining a frame.

Because of the likelihood of confusion between these meanings of "coordinate system," **`coordinates`** avoids this term wherever possible, and instead adopts the following terms (loosely inspired by the IAU2000 resolutions on celestial coordinate systems):

- A "Coordinate Representation" is a particular way of describing a unique

point in a vector space. (Here, this means three-dimensional space, but future extensions might have different dimensionality, particularly if relativistic effects are desired.) Examples include Cartesian coordinates, cylindrical polar, or latitude/longitude spherical polar coordinates. Note that this term applies to the positions, *not* their velocities or other derivatives (which are represented as "differential" classes).
- A "Reference System" is a scheme for orienting points in a space and describing how they transforms to other systems. Examples include the ICRS, equatorial coordinates with mean equinox, or the WGS84 geoid for latitude/longitude on the Earth.
- A "Coordinate Frame," "Reference Frame," or just "Frame" is a specific realization of a reference system (e.g., the ICRF, or J2000 equatorial coordinates). For some systems, there may be only one meaningful frame, while others may have many different frames (differentiated by something like a different equinox, or a different set of reference points).
- A "Coordinate" is a combination of all of the above that specifies a unique point.

Fast In-Place Modification of Coordinates

For some applications the recommended method of Modifying Coordinate Objects In-place may not be fast enough due to the extensive validation performed in that process to ensure correctness. Likewise, you may find that creating another coordinate frame with different data using **`realize_frame`** does not meet your performance requirements.

For these high-performance situations, you can directly modify in-place the representation data in the frame object as shown in this example:

```
>>> import astropy.units as u
>>> from astropy.coordinates import SkyCoord
>>> sc = SkyCoord([1,2],[3,4], unit='deg')
>>> sc.data.lon[()] = [10, 20] * u.deg
>>> sc.data.lat[1] = 40 * u.deg

>>> sc.cache.clear()  # IMPORTANT TO DO THIS!

>>> sc
<SkyCoord (ICRS): (ra, dec) in deg
    [(10., 3.), (20., 40.)]>
```

Notice that the `.data` representation object uses different names for the components than in the coordinate object. If you wish to inspect the mapping between frame attributes (e.g., `.ra`) and representation attributes (e.g., `.lon`) you can look at the following dictionary:

```
>>> sc.representation_component_names
OrderedDict([('ra', 'lon'), ('dec', 'lat'), ('distance',
'distance')])
```

Warning

You *must* include the step to clear the cache as shown. Failing to do so will cause the object to be inconsistent and likely result in incorrect results. `SkyCoord` and `BaseCoordinateFrame` cache various kinds of information for performance reasons, so you need clear the cache so that the new representation values are used when required.

You should note that the only way to modify the data in a frame is by using the `.data` attribute directly and not the aliases for components on the frame. For example the following will *appear* to give a correct result but it does not actually modify the underlying representation data:

```
>>> sc.ra[1] = 20 * u.deg  # THIS IS WRONG
```

This problem is related to the current implementation of performance-based caching and cannot be easily resolved.

In addition, another resource for the capabilities of this package is the `astropy.coordinates.tests.test_api_ape5` testing file. It showcases most of the major capabilities of the package, and hence is a useful supplement to this document. You can see it by either downloading a copy of the Astropy source code, or typing the following in an IPython session:

```
In [1]: from astropy.coordinates.tests import test_api_ape5
In [2]: test_api_ape5??
```

Performance Tips

If you are using `SkyCoord` for many different coordinates, you will see much better performance if you create a single `SkyCoord` with arrays of coordinates as opposed to creating individual `SkyCoord` objects for each individual coordinate:

```
>>> coord = SkyCoord(ra_array, dec_array, unit='deg')
```

In addition, looping over a `SkyCoord` object can be slow. If you need to transform the coordinates to a different frame, it is much faster to transform a single `SkyCoord` with arrays of values as opposed to looping over the `SkyCoord` and transforming them individually.

Finally, for more advanced users, note that you can use broadcasting to transform **SkyCoord** objects into frames with vector properties.

Example

To use broadcasting to transform **SkyCoord** objects into frames with vector properties:

```
>>> from astropy.coordinates import SkyCoord, EarthLocation
>>> from astropy import coordinates as coord
>>> from astropy.coordinates.tests.utils import
randomly_sample_sphere
>>> from astropy.time import Time
>>> from astropy import units as u
>>> import numpy as np

>>> # 1000 random locations on the sky
>>> ra, dec, _ = randomly_sample_sphere(1000)
>>> coos = SkyCoord(ra, dec)

>>> # 300 times over the space of 10 hours
>>> times = Time.now() + np.linspace(-5, 5, 300)*u.hour

>>> # note the use of broadcasting so that 300 times are broadcast
against 1000 positions
>>> lapalma = EarthLocation.from_geocentric(5327448.9957829,
-1718665.73869569, 3051566.90295403, unit='m')
>>> aa_frame = coord.AltAz(obstime=times[:, np.newaxis],
location=lapalma)

>>> # calculate alt-az of each object at each time.
>>> aa_coos = coos.transform_to(aa_frame)
```

Improving Performance for Arrays of `obstime`

The most expensive operations when transforming between observer-dependent coordinate frames (e.g. `AltAz`) and sky-fixed frames (e.g. `ICRS`) are the calculation of the orientation and position of Earth.

If **SkyCoord** instances are transformed for a large number of closely spaced `obstime`, these calculations can be sped up by factors up to 100, whilst still keeping micro-arcsecond precision, by utilizing interpolation instead of calculating Earth orientation parameters for each individual point.

To use interpolation for the astrometric values in coordinate transformation, use:

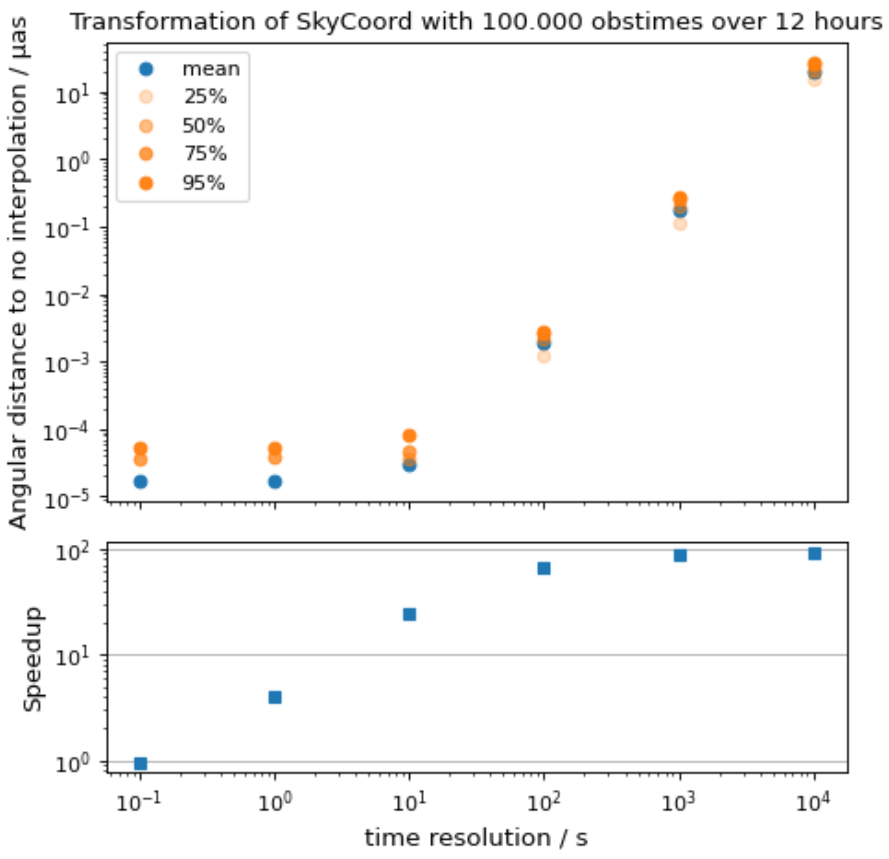
```
>>> from astropy.coordinates import SkyCoord, EarthLocation, AltAz
>>> from astropy.coordinates.erfa_astrom import erfa_astrom,
ErfaAstromInterpolator
>>> from astropy.time import Time
>>> from time import perf_counter
>>> import numpy as np
>>> import astropy.units as u


>>> # array with 10000 obstimes
>>> obstime = Time('2010-01-01T20:00') + np.linspace(0, 6, 10000) *
u.hour
>>> location = location = EarthLocation(lon=-17.89 * u.deg, lat=28.76
* u.deg, height=2200 * u.m)
>>> frame = AltAz(obstime=obstime, location=location)
>>> crab = SkyCoord(ra='05h34m31.94s', dec='22d00m52.2s')

>>> # transform with default transformation and print duration
>>> t0 = perf_counter()
>>> crab_altaz = crab.transform_to(frame)
>>> print(f'Transformation took {perf_counter() - t0:.2f} s')
Transformation took 1.77 s

>>> # transform with interpolating astrometric values
>>> t0 = perf_counter()
>>> with erfa_astrom.set(ErfaAstromInterpolator(300 * u.s)):
...     crab_altaz_interpolated = crab.transform_to(frame)
>>> print(f'Transformation took {perf_counter() - t0:.2f} s')
Transformation took 0.03 s

>>> err = crab_altaz.separation(crab_altaz_interpolated)
>>> print(f'Mean error of interpolation:
{err.to(u.microarcsecond).mean():.4f}')
Mean error of interpolation: 0.0... uarcsec

>>> # To set erfa_astrom for a whole session, use it without context
manager:
>>> erfa_astrom.set(ErfaAstromInterpolator(300 * u.s))
```

Here, we look into choosing an appropriate `time_resolution`. We will transform a single sky coordinate for lots of observation times from `ICRS` to `AltAz` and evaluate precision and runtime for different values for `time_resolution` compared to the non-interpolating, default approach.

```
from time import perf_counter

import numpy as np
import matplotlib.pyplot as plt
```

```
from astropy.coordinates.erfa_astrom import erfa_astrom,
ErfaAstromInterpolator
from astropy.coordinates import SkyCoord, EarthLocation, AltAz
from astropy.time import Time
import astropy.units as u

np.random.seed(1337)

# 100_000 times randomly distributed over 12 hours
t = Time('2020-01-01T20:00:00') + np.random.uniform(0, 1, 10_000) *
u.hour

location = location = EarthLocation(
    lon=-17.89 * u.deg, lat=28.76 * u.deg, height=2200 * u.m
)

# A celestial object in ICRS
crab = SkyCoord.from_name("Crab Nebula")

# target horizontal coordinate frame
altaz = AltAz(obstime=t, location=location)


# the reference transform using no interpolation
t0 = perf_counter()
no_interp = crab.transform_to(altaz)
reference = perf_counter() - t0
print(f'No Interpolation took {reference:.4f} s')


# now the interpolating approach for different time resolutions
resolutions = 10.0**np.arange(-1, 5) * u.s
times = []
seps = []

for resolution in resolutions:
    with erfa_astrom.set(ErfaAstromInterpolator(resolution)):
        t0 = perf_counter()
        interp = crab.transform_to(altaz)
        duration = perf_counter() - t0

    print(
        f'Interpolation with {resolution.value: 9.1f}
{str(resolution.unit)}'
        f' resolution took {duration:.4f} s'
        f' ({reference / duration:5.1f}x faster) '
    )
    seps.append(no_interp.separation(interp))
    times.append(duration)

seps = u.Quantity(seps)

fig = plt.figure()

ax1, ax2 = fig.subplots(2, 1, gridspec_kw={'height_ratios': [2, 1]},
sharex=True)

ax1.plot(
    resolutions.to_value(u.s),
    seps.mean(axis=1).to_value(u.microarcsecond),
    'o', label='mean',
)

for p in [25, 50, 75, 95]:
    ax1.plot(
        resolutions.to_value(u.s),
        np.percentile(seps.to_value(u.microarcsecond), p, axis=1),
        'o', label=f'{p}%', color='C1', alpha=p / 100,
    )

ax1.set_title('Transformation of SkyCoord with 100.000 obstimes over
12 hours')

ax1.legend()
ax1.set_xscale('log')
ax1.set_yscale('log')
ax1.set_ylabel('Angular distance to no interpolation / µas')

ax2.plot(resolutions.to_value(u.s), reference / np.array(times), 's')
ax2.set_yscale('log')
ax2.set_ylabel('Speedup')
ax2.set_xlabel('time resolution / s')

ax2.yaxis.grid()
fig.tight_layout()
```

(png, svg, pdf)



See Also

Some references that are particularly useful in understanding subtleties of the coordinate systems implemented here include:

- USNO Circular 179

  A useful guide to the IAU 2000/2003 work surrounding ICRS/IERS/CIRS and related problems in precision coordinate system work.

- Standards Of Fundamental Astronomy

  The definitive implementation of IAU-defined algorithms. The “SOFA Tools for Earth Attitude” document is particularly valuable for understanding the latest IAU standards in detail.

- IERS Conventions (2010)

  An exhaustive reference covering the ITRS, the IAU2000 celestial coordinates framework, and other related details of modern coordinate conventions.

- Meeus, J. “Astronomical Algorithms”

  A valuable text describing details of a wide range of coordinate-related problems and concepts.

- Revisiting Spacetrack Report #3

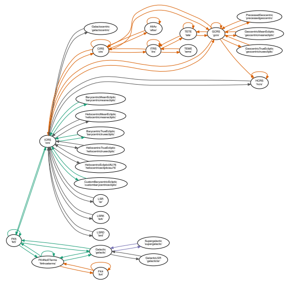
A discussion of the simplified general perturbation (SGP) for satellite orbits, with a description of the True Equator Mean Equinox (TEME) coordinate frame.

Built-in Frame Classes

astropy.coordinates.builtin_frames Package

The diagram below shows all of the built in coordinate systems, their aliases (useful for converting other coordinates to them using attribute-style access) and the pre-defined transformations between them. The user is free to override any of these transformations by defining new transformations between these systems, but the pre-defined transformations should be sufficient for typical usage.

The color of an edge in the graph (i.e. the transformations between two frames) is set by the type of transformation; the legend box defines the mapping from transform class name to color.



AffineTransform: →

FunctionTransform: →

FunctionTransformWithFiniteDifference: →

StaticMatrixTransform: →

DynamicMatrixTransform: →

Classes

`ICRS`(*args[, copy, representation_type, …])	A coordinate or frame in the ICRS system.
`FK5`(*args[, copy, representation_type, …])	A coordinate or frame in the FK5 system.
`FK4`(*args[, copy, representation_type, …])	A coordinate or frame in the FK4 system.
`FK4NoETerms`(*args[, copy, …])	A coordinate or frame in the FK4 system, but with the E-terms of aberration removed.
`Galactic`(*args[, copy, representation_type, …])	A coordinate or frame in the Galactic coordinate system.
`Galactocentric`(*args, **kwargs)	A coordinate or frame in the Galactocentric system.

`galactocentric_frame_defaults`()	This class controls the global setting of default values for the frame attributes in the `Galactocentric` frame, which may be updated in future versions of `astropy`.
`Supergalactic`(*args[, copy, …])	Supergalactic Coordinates (see Lahav et al. 2000, <https://ui.adsabs.harvard.edu/abs/2000MNRAS.312..166L>, and references therein).
`AltAz`(*args, **kwargs)	A coordinate or frame in the Altitude-Azimuth system (Horizontal coordinates).
`GCRS`(*args[, copy, representation_type, …])	A coordinate or frame in the Geocentric Celestial Reference System (GCRS).
`CIRS`(*args[, copy, representation_type, …])	A coordinate or frame in the Celestial Intermediate Reference System (CIRS).
`ITRS`(*args[, copy, representation_type, …])	A coordinate or frame in the International Terrestrial Reference System (ITRS).
`HCRS`(*args[, copy, representation_type, …])	A coordinate or frame in a Heliocentric system, with axes aligned to ICRS.
`TEME`(*args[, copy, representation_type, …])	A coordinate or frame in the True Equator Mean Equinox frame (TEME).
`TETE`(*args[, copy, representation_type, …])	An equatorial coordinate or frame using the True Equator and True Equinox (TETE).
`PrecessedGeocentric`(*args[, copy, …])	A coordinate frame defined in a similar manner as GCRS, but precessed to a requested (mean) equinox.
`GeocentricMeanEcliptic`(*args[, copy, …])	Geocentric mean ecliptic coordinates.
`BarycentricMeanEcliptic`(*args[, copy, …])	Barycentric mean ecliptic coordinates.
`HeliocentricMeanEcliptic`(*args[, copy, …])	Heliocentric mean ecliptic coordinates.
`GeocentricTrueEcliptic`(*args[, copy, …])	Geocentric true ecliptic coordinates.
`BarycentricTrueEcliptic`(*args[, copy, …])	Barycentric true ecliptic coordinates.
`HeliocentricTrueEcliptic`(*args[, copy, …])	Heliocentric true ecliptic coordinates.
`SkyOffsetFrame`(*args, **kwargs)	A frame which is relative to some specific position and oriented to match its frame.
`GalacticLSR`(*args[, copy, …])	A coordinate or frame in the Local Standard of Rest (LSR), axis-aligned to the `Galactic` frame.
`LSR`(*args[, copy, representation_type, …])	A coordinate or frame in the Local Standard of Rest (LSR).
`LSRK`(*args[, copy, representation_type, …])	A coordinate or frame in the Kinematic Local Standard of Rest (LSR).
`LSRD`(*args[, copy, representation_type, …])	A coordinate or frame in the Dynamical Local Standard of Rest (LSRD)
`BaseEclipticFrame`(*args[, copy, …])	A base class for frames that have names and conventions like that of ecliptic frames.

BaseRADecFrame(*args[, copy, …])	A base class that defines default representation info for frames that represent longitude and latitude as Right Ascension and Declination following typical “equatorial” conventions.
HeliocentricEclipticIAU76(*args[, copy, …])	Heliocentric mean (IAU 1976) ecliptic coordinates.
CustomBarycentricEcliptic(*args[, copy, …])	Barycentric ecliptic coordinates with custom obliquity.

Reference/API

astropy.coordinates Package

This subpackage contains classes and functions for celestial coordinates of astronomical objects. It also contains a framework for conversions between coordinate systems.

Functions

cartesian_to_spherical(x, y, z)	Converts 3D rectangular cartesian coordinates to spherical polar coordinates.
concatenate(coords)	Combine multiple coordinate objects into a single SkyCoord.
concatenate_representations(reps)	Combine multiple representation objects into a single instance by concatenating the data in each component.
get_body(body, time[, location, ephemeris])	Get a SkyCoord for a solar system body as observed from a location on Earth in the GCRS reference system.
get_body_barycentric(body, time[, ephemeris])	Calculate the barycentric position of a solar system body.
get_body_barycentric_posvel(body, time[, …])	Calculate the barycentric position and velocity of a solar system body.
get_constellation(coord[, short_name, …])	Determines the constellation(s) a given coordinate object contains.
get_icrs_coordinates(name[, parse, cache])	Retrieve an ICRS object by using an online name resolving service to retrieve coordinates for the specified name.
get_moon(time[, location, ephemeris])	Get a SkyCoord for the Earth’s Moon as observed from a location on Earth in the GCRS reference system.
get_sun(time)	Determines the location of the sun at a given time (or times, if the input is an array Time object), in geocentric coordinates.
make_transform_graph_docs(transform_graph)	Generates a string that can be used in other docstrings to include a transformation graph, showing the available transforms and coordinate systems.
match_coordinates_3d(matchcoord, catalogcoord)	Finds the nearest 3-dimensional matches of a coordinate or coordinates in a set of catalog coordinates.

match_coordinates_sky(matchcoord, catalogcoord)	Finds the nearest on-sky matches of a coordinate or coordinates in a set of catalog coordinates.
search_around_3d(coords1, coords2, distlimit)	Searches for pairs of points that are at least as close as a specified distance in 3D space.
search_around_sky(coords1, coords2, seplimit)	Searches for pairs of points that have an angular separation at least as close as a specified angle.
spherical_to_cartesian(r, lat, lon)	Converts spherical polar coordinates to rectangular cartesian coordinates.

Classes

AffineTransform(transform_func, fromsys, tosys)	A coordinate transformation specified as a function that yields a 3 x 3 cartesian transformation matrix and a tuple of displacement vectors.
AltAz(*args, **kwargs)	A coordinate or frame in the Altitude-Azimuth system (Horizontal coordinates).
Angle(angle[, unit, dtype, copy])	One or more angular value(s) with units equivalent to radians or degrees.
Attribute([default, secondary_attribute])	A non-mutable data descriptor to hold a frame attribute.
BarycentricMeanEcliptic(*args[, copy, …])	Barycentric mean ecliptic coordinates.
BarycentricTrueEcliptic(*args[, copy, …])	Barycentric true ecliptic coordinates.
BaseAffineTransform(fromsys, tosys[, …])	Base class for common functionality between the `AffineTransform` -type subclasses.
BaseCoordinateFrame(*args[, copy, …])	The base class for coordinate frames.
BaseDifferential(*args, **kwargs)	A base class representing differentials of representations.
BaseEclipticFrame(*args[, copy, …])	A base class for frames that have names and conventions like that of ecliptic frames.
BaseRADecFrame(*args[, copy, …])	A base class that defines default representation info for frames that represent longitude and latitude as Right Ascension and Declination following typical “equatorial” conventions.
BaseRepresentation(*args[, differentials])	Base for representing a point in a 3D coordinate system.
BaseRepresentationOrDifferential(*args, **kwargs)	3D coordinate representations and differentials.
BaseSphericalCosLatDifferential(*args, **kwargs)	Differentials from points on a spherical base representation.
BaseSphericalDifferential(*args, **kwargs)	

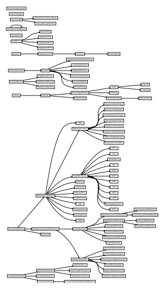
`BoundsError`	Raised when an angle is outside of its user-specified bounds.
`CIRS`(*args[, copy, representation_type, …])	A coordinate or frame in the Celestial Intermediate Reference System (CIRS).
`CartesianDifferential`(d_x[, d_y, d_z, unit, …])	Differentials in of points in 3D cartesian coordinates.
`CartesianRepresentation`(x[, y, z, unit, …])	Representation of points in 3D cartesian coordinates.
`CartesianRepresentationAttribute`([default, …])	A frame attribute that is a CartesianRepresentation with specified units.
`CompositeTransform`(transforms, fromsys, tosys)	A transformation constructed by combining together a series of single-step transformations.
`ConvertError`	Raised if a coordinate system cannot be converted to another
`CoordinateAttribute`(frame[, default, …])	A frame attribute which is a coordinate object.
`CoordinateTransform`(fromsys, tosys[, …])	An object that transforms a coordinate from one system to another.
`CustomBarycentricEcliptic`(*args[, copy, …])	Barycentric ecliptic coordinates with custom obliquity.
`CylindricalDifferential`(d_rho[, d_phi, d_z, …])	Differential(s) of points in cylindrical coordinates.
`CylindricalRepresentation`(rho[, phi, z, …])	Representation of points in 3D cylindrical coordinates.
`DifferentialAttribute`([default, …])	A frame attribute which is a differential instance.
`Distance`([value, unit, z, cosmology, …])	A one-dimensional distance.
`DynamicMatrixTransform`(matrix_func, fromsys, …)	A coordinate transformation specified as a function that yields a 3 x 3 cartesian transformation matrix.
`EarthLocation`(*args, **kwargs)	Location on the Earth.
`EarthLocationAttribute`([default, …])	A frame attribute that can act as a `EarthLocation`.
`FK4`(*args[, copy, representation_type, …])	A coordinate or frame in the FK4 system.
`FK4NoETerms`(*args[, copy, …])	A coordinate or frame in the FK4 system, but with the E-terms of aberration removed.
`FK5`(*args[, copy, representation_type, …])	A coordinate or frame in the FK5 system.
`FunctionTransform`(func, fromsys, tosys[, …])	A coordinate transformation defined by a function that accepts a coordinate object and returns the transformed coordinate object.
`FunctionTransformWithFiniteDifference`(func, …)	A coordinate transformation that works like a `FunctionTransform`, but computes velocity shifts based on the finite-difference relative to one of the frame attributes.
`GCRS`(*args[, copy, representation_type, …])	A coordinate or frame in the Geocentric Celestial Reference System (GCRS).
`Galactic`(*args[, copy, representation_type, …])	A coordinate or frame in the Galactic coordinate system.
`GalacticLSR`(*args[, copy, …])	A coordinate or frame in the Local Standard of Rest (LSR), axis-aligned to the `Galactic` frame.

Galactocentric(*args, **kwargs)	A coordinate or frame in the Galactocentric system.
GenericFrame(frame_attrs)	A frame object that can't store data but can hold any arbitrary frame attributes.
GeocentricMeanEcliptic(*args[, copy, …])	Geocentric mean ecliptic coordinates.
GeocentricTrueEcliptic(*args[, copy, …])	Geocentric true ecliptic coordinates.
HCRS(*args[, copy, representation_type, …])	A coordinate or frame in a Heliocentric system, with axes aligned to ICRS.
HeliocentricEclipticIAU76(*args[, copy, …])	Heliocentric mean (IAU 1976) ecliptic coordinates.
HeliocentricMeanEcliptic(*args[, copy, …])	Heliocentric mean ecliptic coordinates.
HeliocentricTrueEcliptic(*args[, copy, …])	Heliocentric true ecliptic coordinates.
ICRS(*args[, copy, representation_type, …])	A coordinate or frame in the ICRS system.
ITRS(*args[, copy, representation_type, …])	A coordinate or frame in the International Terrestrial Reference System (ITRS).
IllegalHourError(hour)	Raised when an hour value is not in the range [0,24).
IllegalHourWarning(hour[, alternativeactionstr])	Raised when an hour value is 24.
IllegalMinuteError(minute)	Raised when an minute value is not in the range [0,60].
IllegalMinuteWarning(minute[, …])	Raised when a minute value is 60.
IllegalSecondError(second)	Raised when an second value (time) is not in the range [0,60].
IllegalSecondWarning(second[, …])	Raised when a second value is 60.
LSR(*args[, copy, representation_type, …])	A coordinate or frame in the Local Standard of Rest (LSR).
LSRD(*args[, copy, representation_type, …])	A coordinate or frame in the Dynamical Local Standard of Rest (LSRD)
LSRK(*args[, copy, representation_type, …])	A coordinate or frame in the Kinematic Local Standard of Rest (LSR).
Latitude(angle[, unit])	Latitude-like angle(s) which must be in the range -90 to +90 deg.
Longitude(angle[, unit, wrap_angle])	Longitude-like angle(s) which are wrapped within a contiguous 360 degree range.
PhysicsSphericalDifferential(d_phi[, …])	Differential(s) of 3D spherical coordinates using physics convention.
PhysicsSphericalRepresentation(phi[, theta, …])	Representation of points in 3D spherical coordinates (using the physics convention of using `phi` and `theta` for azimuth and inclination from the pole).
PrecessedGeocentric(*args[, copy, …])	A coordinate frame defined in a similar manner as GCRS, but precessed to a requested (mean) equinox.

`QuantityAttribute`([default, …])	A frame attribute that is a quantity with specified units and shape (optionally).
`RadialDifferential`(*args, **kwargs)	Differential(s) of radial distances.
`RadialRepresentation`(distance[, …])	Representation of the distance of points from the origin.
`RangeError`	Raised when some part of an angle is out of its valid range.
`RepresentationMapping`(reprname, framename[, …])	This `namedtuple` is used with the `frame_specific_representation_info` attribute to tell frames what attribute names (and default units) to use for a particular representation.
`SkyCoord`(*args[, copy])	High-level object providing a flexible interface for celestial coordinate representation, manipulation, and transformation between systems.
`SkyCoordInfo`([bound])	Container for meta information like name, description, format.
`SkyOffsetFrame`(*args, **kwargs)	A frame which is relative to some specific position and oriented to match its frame.
`SpectralCoord`(value[, unit, observer, …])	A spectral coordinate with its corresponding unit.
`SpectralQuantity`(value[, unit, …])	One or more value(s) with spectral units.
`SphericalCosLatDifferential`(d_lon_coslat[, …])	Differential(s) of points in 3D spherical coordinates.
`SphericalDifferential`(d_lon[, d_lat, …])	Differential(s) of points in 3D spherical coordinates.
`SphericalRepresentation`(lon[, lat, …])	Representation of points in 3D spherical coordinates.
`StaticMatrixTransform`(matrix, fromsys, tosys)	A coordinate transformation defined as a 3 x 3 cartesian transformation matrix.
`Supergalactic`(*args[, copy, …])	Supergalactic Coordinates (see Lahav et al. 2000, <https://ui.adsabs.harvard.edu/abs/2000MNRAS.312..166L>, and references therein).
`TEME`(*args[, copy, representation_type, …])	A coordinate or frame in the True Equator Mean Equinox frame (TEME).
`TETE`(*args[, copy, representation_type, …])	An equatorial coordinate or frame using the True Equator and True Equinox (TETE).
`TimeAttribute`([default, secondary_attribute])	Frame attribute descriptor for quantities that are Time objects.
`TransformGraph`()	A graph representing the paths between coordinate frames.
`UnitSphericalCosLatDifferential`(d_lon_coslat)	Differential(s) of points on a unit sphere.
`UnitSphericalDifferential`(d_lon[, d_lat, copy])	Differential(s) of points on a unit sphere.
`UnitSphericalRepresentation`(lon[, lat, …])	Representation of points on a unit sphere.
`UnknownSiteException`(site, attribute[, …])	

`galactocentric_frame_defaults`()	This class controls the global setting of default values for the frame attributes in the `Galactocentric` frame, which may be updated in future versions of `astropy`.
`solar_system_ephemeris`()	Default ephemerides for calculating positions of Solar-System bodies.

Class Inheritance Diagram



World Coordinate System (`astropy.wcs`)

Introduction

World Coordinate Systems (WCSs) describe the geometric transformations between one set of coordinates and another. A common application is to map the pixels in an image onto the celestial sphere. Another common application is to map pixels to wavelength in a spectrum.

astropy.wcs contains utilities for managing World Coordinate System (WCS) transformations defined in several elaborate FITS WCS standard conventions. These transformations work both forward (from pixel to world) and backward (from world to pixel).

For historical reasons and to support legacy software, **`astropy.wcs`** maintains two separate application interfaces. The `High-Level API` should be used by most applications. It abstracts out the underlying object and works transparently with other packages which support the Common Python Interface for WCS, allowing for a more flexible approach to the problem and avoiding the limitations of the FITS WCS standard.

The `Low Level API` is the original **`astropy.wcs`** API. It ties applications to the **`astropy.wcs`** package and limits the transformations to the three distinct types supported by it:

- Core WCS, as defined in the FITS WCS standard, based on Mark Calabretta's wcslib. (Also includes `TPV` and `TPD` distortion, but not `SIP`).

- Simple Imaging Polynomial (SIP) convention. (See note about SIP in headers.)
- Table lookup distortions as defined in the FITS WCS distortion paper.

Pixel Conventions and Definitions

Both APIs assume that integer pixel values fall at the center of pixels (as assumed in the FITS WCS standard, see Section 2.1.4 of Greisen et al., 2002, A&A 446, 747).

However, there's a difference in what is considered to be the first pixel. The `High Level API` follows the Python and C convention that the first pixel is the 0-th one, i.e. the first pixel spans pixel values -0.5 to + 0.5. The `Low Level API` takes an additional `origin` argument with values of 0 or 1 indicating whether the input arrays are 0- or 1-based. The Low-level interface assumes Cartesian order (x, y) of the input coordinates, however the Common Interface for World Coordinate System accepts both conventions. The order of the pixel coordinates ((x, y) vs (row, column)) in the Common API depends on the method or property used, and this can normally be determined from the property or method name. Properties and methods containing "pixel" assume (x, y) ordering, while properties and methods containing "array" assume (row, column) ordering.

A Simple Example

One example of the use of the high-level WCS API is to use the **`pixel_to_world`** to yield the simplest WCS with default values, converting from pixel to world coordinates:

```
>>> from astropy.io import fits
>>> from astropy.wcs import WCS
>>> from astropy.utils.data import get_pkg_data_filename
>>> fn = get_pkg_data_filename('data/j94f05bgq_flt.fits',
package='astropy.wcs.tests')
>>> f = fits.open(fn)
>>> w = WCS(f[1].header)
>>> sky = w.pixel_to_world(30, 40)
>>> print(sky)
<SkyCoord (ICRS): (ra, dec) in deg
    (5.52844243, -72.05207809)>
```

Similarly, another use of the high-level API is to use the **`world_to_pixel`** to yield another simple WCS, while converting from world to pixel coordinates:

```
>>> from astropy.io import fits
>>> from astropy.wcs import WCS
>>> from astropy.utils.data import get_pkg_data_filename
>>> fn = get_pkg_data_filename('data/j94f05bgq_flt.fits',
package='astropy.wcs.tests')
>>> f = fits.open(fn)
>>> w = WCS(f[1].header)
>>> x, y = w.world_to_pixel(sky)
>>> print(x, y)
30.00000214673885 39.999999958235094
```

Using `astropy.wcs`

Shared Python Interface for World Coordinate Systems

Background

The **`WCS`** class implements what is considered the most common ‘standard’ for representing world coordinate systems in FITS files, but it cannot represent arbitrarily complex transformations and there is no agreement on how to use the standard beyond FITS files. Therefore, other world coordinate system transformation approaches exist, such as the gwcs package being developed for the James Webb Space Telescope (which is also applicable to other data).

Since one of the goals of the Astropy Project is to improve interoperability between packages, we have collaboratively defined a standardized application programming interface (API) for world coordinate system objects to be used in Python. This API is described in the Astropy Proposal for Enhancements (APE) 14: A shared Python interface for World Coordinate Systems.

The core astropy package provides base classes that define the low- and high-level APIs described in APE 14 in the **`astropy.wcs.wcsapi`** module, and these are listed in the Acknowledgments and Licenses section below.

Overview

While the full details and motivation for the API are detailed in APE 14, this documentation summarizes the elements that are implemented directly in the astropy core package. The high-level interface is likely of most interest to the average user. In particular, the most important methods are the **`pixel_to_world()`** and **`world_to_pixel()`** methods. These provide the essential elements of WCS: mapping to and from world coordinates. The remainder generally provide information about the *kind* of world coordinates or similar information about the structure of the WCS.

In a bit more detail, the key classes implemented here are a high-level that provides the main user interface (**BaseHighLevelWCS** and subclasses), and a lower-level interface (**BaseLowLevelWCS** and subclasses). These can be distinct objects *or* the same one. For FITS-WCS, the **WCS** object meant for FITS-WCS follows both interfaces, allowing immediate use of this API with files that already contain FITS-WCS. More concrete examples are outlined below.

Basic usage

Let's start off by looking at the shared Python interface for WCS by using a simple image with two celestial axes (Right Ascension and Declination):

```
>>> from astropy.wcs import WCS
>>> from astropy.utils.data import get_pkg_data_filename
>>> from astropy.io import fits
>>> filename =
get_pkg_data_filename('galactic_center/gc_2mass_k.fits')
>>> hdu = fits.open(filename)[0]
>>> wcs = WCS(hdu.header)
>>> wcs
WCS Keywords
Number of WCS axes: 2
CTYPE : 'RA---TAN'  'DEC--TAN'
CRVAL : 266.4  -28.93333
CRPIX : 361.0  360.5
NAXIS : 721  720
```

We can check how many pixel and world axes are in the transformation as well as the shape of the data the WCS applies to:

```
>>> wcs.pixel_n_dim
2
>>> wcs.world_n_dim
2
>>> wcs.array_shape
(720, 721)
```

Note that the array shape should match that of the data:

```
>>> hdu.data.shape
(720, 721)
```

As mentioned in Pixel Conventions and Definitions, what would normally be considered the 'y-axis' of the image (when looking at it visually) is the first dimension, while the 'x-axis' of the image is the second dimension. Thus **`array_shape`** returns the shape in the *opposite* order to the NAXIS keywords in the FITS header (in the case of FITS-WCS). If you are interested in the data shape in the reverse order (which would match the NAXIS order in the case of FITS-WCS), then you can use **`pixel_shape`**:

```
>>> wcs.pixel_shape
(721, 720)
```

Let's now check what the physical type of each axis is:

```
>>> wcs.world_axis_physical_types
['pos.eq.ra', 'pos.eq.dec']
```

This is indeed an image with two celestial axes.

The main part of the new interface defines standard methods for transforming coordinates. The most convenient way is to use the high-level methods **`pixel_to_world()`** and **`world_to_pixel()`**, which can transform directly to astropy objects:

```
>>> coord = wcs.pixel_to_world([1, 2], [4, 3])
>>> coord
<SkyCoord (FK5: equinox=2000.0): (ra, dec) in deg
    [(266.97242993, -29.42584415), (266.97084321, -29.42723968)]>
```

Similarly, we can transform astropy objects back - we can test this by creating Galactic coordinates and these will automatically be converted:

```
>>> from astropy.coordinates import SkyCoord
>>> coord = SkyCoord('00h00m00s +00d00m00s', frame='galactic')
>>> pixels = wcs.world_to_pixel(coord)
>>> pixels
(array(356.85179997), array(357.45340331))
```

If you are looking to index the original data using these pixel coordinates, be sure to instead use **`world_to_array_index()`** which returns the coordinates in the correct order to index Numpy arrays, and also rounds to the nearest integer values:

```
>>> index = wcs.world_to_array_index(coord)
>>> index
(357, 357)
```

```
>>> hdu.data[index]
563.7532
```

Advanced usage

Let’s now take a look at a WCS for a spectral cube (two celestial axes and one spectral axis):

```
>>> filename = get_pkg_data_filename('l1448/l1448_13co.fits')
>>> hdu = fits.open(filename)[0]
>>> wcs = WCS(hdu.header)
>>> wcs
WCS Keywords
Number of WCS axes: 3
CTYPE : 'RA---SFL'  'DEC--SFL'  'VOPT'
CRVAL : 57.6599999999  0.0  -9959.44378305
CRPIX : -799.0  -4741.913  -187.0
PC1_1 PC1_2 PC1_3  : 1.0  0.0  0.0
PC2_1 PC2_2 PC2_3  : 0.0  1.0  0.0
PC3_1 PC3_2 PC3_3  : 0.0  0.0  1.0
CDELT : -0.006388889  0.006388889  66.42361
NAXIS : 105  105  53
```

As before we can check how many pixel and world axes are in the transformation as well as the shape of the data the WCS applies to, as well as the physical types of each axis:

```
>>> wcs.pixel_n_dim
3
>>> wcs.world_n_dim
3
>>> wcs.array_shape
(53, 105, 105)
>>> wcs.world_axis_physical_types
['pos.eq.ra', 'pos.eq.dec', 'spect.dopplerVeloc.opt']
```

This is indeed a spectral cube, with RA/Dec and a velocity axis.

As before, we can convert between pixels and high-level Astropy objects:

```
>>> celestial, spectral = wcs.pixel_to_world([1, 2], [4, 3], [2, 3])
>>> celestial
<SkyCoord (ICRS): (ra, dec) in deg
    [(51.73115731, 30.32750025), (51.72414268, 30.32111136)]>
>>> spectral
```

```
<SpectralCoord
   (target: <ICRS Coordinate: (ra, dec, distance) in (deg, deg, kpc)
                (57.66, 0., 1000.)
             (pm_ra_cosdec, pm_dec, radial_velocity) in (mas / yr,
mas / yr, km / s)
                (0., 0., 0.)>)
  [2661.04211695, 2727.46572695] m / s>
```

and back:

```
>>> from astropy import units as u
>>> coord = SkyCoord('03h26m36.4901s +30d45m22.2012s')
>>> pixels = wcs.world_to_pixel(coord, 3000 * u.m / u.s)
>>> pixels
(array(8.11341207), array(71.0956641), array(7.10297292))
```

And as before we can index array values using:

```
>>> index = wcs.world_to_array_index(coord, 3000 * u.m / u.s)
>>> index
(7, 71, 8)
>>> hdu.data[index]
0.22262384
```

If you are interested in converting to/from world values as simple Python scalars or Numpy arrays without using high-level astropy objects, there are methods such as **`pixel_to_world_values()`** to do this - see Acknowledgments and Licenses section for more details.

Extending the physical types in FITS-WCS

As shown above, the **`world_axis_physical_types`** property returns the list of physical types for each axis. For FITS-WCS, this is determined from the CTYPE values in the header. In cases where the physical type is not known, **`None`** is returned. However, it is possible to override the physical types returned by using the **`custom_ctype_to_ucd_mapping`** context manager. Consider a WCS with the following CTYPE:

```
>>> from astropy.wcs import WCS
>>> wcs = WCS(naxis=1)
>>> wcs.wcs.ctype = ['SPAM']
>>> wcs.world_axis_physical_types
[None]
```

We can specify that for this CTYPE, the physical type should be `'food.spam'`:

```
>>> from astropy.wcs.wcsapi.fitswcs import
custom_ctype_to_ucd_mapping
>>> with custom_ctype_to_ucd_mapping({'SPAM': 'food.spam'}):
...     wcs.world_axis_physical_types
['food.spam']
```

Slicing of WCS objects

A common operation when dealing with data with WCS information attached is to slice the WCS - this can be either to extract the WCS for a sub-region of the data, preserving the overall number of dimensions (e.g. a cutout from an image) or it can be reducing the dimensionality of the data and associated WCS (e.g. extracting a slice from a spectral cube).

The **`SlicedLowLevelWCS`** class can be used to slice any WCS object that conforms to the **`BaseLowLevelWCS`** API. To demonstrate this, let's start off by reading in a spectral cube file:

```
>>> filename = get_pkg_data_filename('l1448/l1448_13co.fits')
>>> wcs = WCS(fits.getheader(filename, ext=0))
```

The `wcs` object is an instance of **`WCS`** which conforms to the **`BaseLowLevelWCS`** API. We can then use the **`SlicedLowLevelWCS`** class to slice the cube:

```
>>> from astropy.wcs.wcsapi import SlicedLowLevelWCS
>>> slices = [10, slice(30, 100), slice(30, 100)]
>>> subwcs = SlicedLowLevelWCS(wcs, slices=slices)
```

The `slices` argument takes any combination of slices, integer values, and ellipsis which would normally slice a Numpy array. In the above case, we are extracting a spectral slice, and in that slice we are extracting a sub-region on the sky.

If you are implementing your own WCS class, you could choose to implement `__getitem__` and have it internally use **`SlicedLowLevelWCS`**. In fact, the **`WCS`** class does this - the example above can be written more succinctly as:

```
>>> wcs[10, 30:100, 30:100]
```

```
<...>
SlicedFITSWCS Transformation

This transformation has 2 pixel and 2 world dimensions

Array shape (Numpy order): (70, 70)

Pixel Dim  Axis Name  Data size  Bounds
        0  None              70  None
        1  None              70  None

World Dim  Axis Name  Physical Type  Units
        0  None       pos.eq.ra      deg
        1  None       pos.eq.dec     deg

Correlation between pixel and world axes:

           Pixel Dim
World Dim    0    1
        0  yes  yes
        1  yes  yes
```

This slicing infrastructure is able to deal with slicing of WCS objects which have correlated axes - in this case, you may end up with a WCS that has a different number of pixel and world coordinates. For example, if we slice a spectral cube to extract a 1D dataset corresponding to a row in the image plane of a spectral slice, the final WCS will have one pixel dimension and two world dimensions (since both RA/Dec vary over the extracted 1D slice):

```
>>> wcs[10, 40, :]
<...>
SlicedFITSWCS Transformation

This transformation has 1 pixel and 2 world dimensions

Array shape (Numpy order): (105,)

Pixel Dim  Axis Name  Data size  Bounds
        0  None             105  None

World Dim  Axis Name  Physical Type  Units
        0  None       pos.eq.ra      deg
        1  None       pos.eq.dec     deg

Correlation between pixel and world axes:

           Pixel Dim
World Dim    0
```

```
         0  yes
         1  yes
```

Legacy Interface

astropy.wcs API

The `Low Level API` or `Legacy Interface` is the original **`astropy.wcs`** API. It supports three types of transforms:

- Core WCS, as defined in the FITS WCS standard, based on Mark Calabretta's wcslib. (Also includes `TPV` and `TPD` distortion, but not `SIP`).
- Simple Imaging Polynomial (SIP) convention. (See note about SIP in headers.)
- Table lookup distortions as defined in the FITS WCS distortion paper.

Each of these transformations can be used independently or together in a standard pipeline. All methods support scalar and array inputs. Note, that all methods require an additional positional argument which is the `origin` of the inputs. It has two possible values - `0` - for zero-based coordinates like numpy arrays or `1` - for 1-based coordinates, like the FITS standard, or those coming from ds9.

The basic workflow is to create a WCS object calling the WCS constructor with an **`Header`** and/or **`HDUList`** object and calling one of the methods below:

```
>>> from astropy import wcs
>>> from astropy.io import fits
>>> from astropy.utils.data import get_pkg_data_filename
>>> fn = get_pkg_data_filename('data/j94f05bgq_flt.fits',
package='astropy.wcs.tests')
>>> f = fits.open(fn)
>>> wcsobj = wcs.WCS(f[1].header)
```

Optionally, if the FITS file uses any deprecated or non-standard features, you may need to call one of the **`fix`** methods on the object.

Use one of the following transformation methods.

1. Between pixels and world coordinates using all distortions:

    - **`all_pix2world`**: Perform all three transformations in series (core WCS, SIP and table lookup distortions) from pixel to world coordinates. Use this one if you're not sure which to use.

```
>>> lon, lat = wcsobj.all_pix2world(30, 40, 0)
>>> print(lon, lat)
5.528442425094046 -72.05207808966726
```

- `all_world2pix`: Perform all three transformations (core WCS, SIP and table lookup distortions) from world to pixel coordinates, using an iterative method if necessary.

  ```
  >>> x, y = wcsobj.all_world2pix(lon, lat, 0)
  >>> print(x, y) #
  30.00000214673885 39.999999958235094
  ```

2. Performing SIP transformations only:

   - `sip_pix2foc`: Convert from pixel to focal plane coordinates using the SIP polynomial coefficients.

     ```
     >>> xsip, ysip = wcsobj.sip_pix2foc(30, 40, 0)
     >>> print(xsip, ysip)
     -1985.8600487630586 -984.4223711273145
     ```

   - `sip_foc2pix`: Convert from focal plane to pixel coordinates using the SIP polynomial coefficients. Note that this method only works if the inverse SIP distortion is specified in the header.

3. Performing distortion paper transformations only:

   - `p4_pix2foc`: Convert from pixel to focal plane coordinates using the table lookup distortion method described in the FITS WCS distortion paper.

   - `det2im`: Convert from detector coordinates to image coordinates. Commonly used for narrow column correction.

Core wcslib API

The core wcslib API supports the FITS WCS standard defined in WCS papers, I, II, III, IV. Note that distortions are not applied if the functions in the core library are used.

1. From pixels to world coordinates:

   - **wcs_pix2world**: Perform just the core WCS transformation from pixel to world coordinates.

   ```
   >>> lon, lat = wcsobj.wcs_pix2world(30, 40, 0)
   >>> print(lon, lat)
   5.527103615238458 -72.0522441352217
   ```

2. From world to pixel coordinates:

   - **wcs_world2pix**: Perform the core WCS transformation from world to pixel coordinates.

   ```
   >>> x, y = wcsobj.wcs_world2pix(lon, lat, 0)
   >>> print(x, y)
   30.000000000223267 40.0000000003696
   ```

Supported projections

As **astropy.wcs** is based on wcslib, it supports the standard projections defined in the FITS WCS standard. These projection codes are three letter strings specified in the second part of the `CTYPEn` keywords (accessible through **Wcsprm.ctype**). For example, a tangent projection with RA, DEC coordinates is defined by `CTYPE1 = RA---TAN` and `CTYPE2 = DEC--TAN`. If a SIP distortion is present the keywords become `CTYPE1 = RA---TAN-SIP` and `CTYPE2 = DEC--TAN-SIP`.

The supported projection codes are:

- `AZP` : zenithal/azimuthal perspective
- `SZP` : slant zenithal perspective
- `TAN` : gnomonic
- `STG` : stereographic
- `SIN` : orthographic/synthesis
- `ARC` : zenithal/azimuthal equidistant
- `ZPN` : zenithal/azimuthal polynomial
- `ZEA` : zenithal/azimuthal equal area
- `AIR` : Airy’s projection
- `CYP` : cylindrical perspective

- `CEA` : cylindrical equal area
- `CAR` : plate carrée
- `MER` : Mercator's projection
- `COP` : conic perspective
- `COE` : conic equal area
- `COD` : conic equidistant
- `COO` : conic orthomorphic
- `SFL` : Sanson-Flamsteed ("global sinusoid")
- `PAR` : parabolic
- `MOL` : Mollweide's projection
- `AIT` : Hammer-Aitoff
- `BON` : Bonne's projection
- `PCO` : polyconic
- `TSC` : tangential spherical cube
- `CSC` : COBE quadrilateralized spherical cube
- `QSC` : quadrilateralized spherical cube
- `HPX` : HEALPix
- `XPH` : HEALPix polar, aka "butterfly"

And, if built with wcslib 5.0 or later, the following polynomial distortions are supported:

- `TPV` : Polynomial distortion
- `TUV` : Polynomial distortion

Note

Though wcslib 5.4 and later handles `SIP` polynomial distortion, for backward compatibility, `SIP` is handled by astropy itself and methods exist to handle it specially.

Examples creating a WCS programmatically

First Example

This example, rather than starting from a FITS header, sets WCS values programmatically, uses those settings to transform some points, and then saves those settings to a new FITS header.

```
# Set the WCS information manually by setting properties of the WCS
# object.

import numpy as np
from astropy import wcs
from astropy.io import fits

# Create a new WCS object.  The number of axes must be set
# from the start
w = wcs.WCS(naxis=2)

# Set up an "Airy's zenithal" projection
# Vector properties may be set with Python lists, or Numpy arrays
w.wcs.crpix = [-234.75, 8.3393]
w.wcs.cdelt = np.array([-0.066667, 0.066667])
w.wcs.crval = [0, -90]
w.wcs.ctype = ["RA---AIR", "DEC--AIR"]
w.wcs.set_pv([(2, 1, 45.0)])

# Three pixel coordinates of interest.
# The pixel coordinates are pairs of [X, Y].
# The "origin" argument indicates whether the input coordinates
# are 0-based (as in Numpy arrays) or
# 1-based (as in the FITS convention, for example coordinates
# coming from DS9).
pixcrd = np.array([[0, 0], [24, 38], [45, 98]], dtype=np.float64)

# Convert pixel coordinates to world coordinates.
# The second argument is "origin" -- in this case we're declaring we
# have 0-based (Numpy-like) coordinates.
world = w.wcs_pix2world(pixcrd, 0)
print(world)

# Convert the same coordinates back to pixel coordinates.
pixcrd2 = w.wcs_world2pix(world, 0)
print(pixcrd2)

# These should be the same as the original pixel coordinates, modulo
# some floating-point error.
assert np.max(np.abs(pixcrd - pixcrd2)) < 1e-6

# The example below illustrates the use of "origin" to convert
between
# 0- and 1- based coordinates when executing the forward and backward
# WCS transform.
x = 0
y = 0
origin = 0
assert (w.wcs_pix2world(x, y, origin) ==
        w.wcs_pix2world(x + 1, y + 1, origin + 1))

# Now, write out the WCS object as a FITS header
header = w.to_header()
```

```
# header is an astropy.io.fits.Header object.  We can use it to
create a new
# PrimaryHDU and write it to a file.
hdu = fits.PrimaryHDU(header=header)
# Save to FITS file
# hdu.writeto('test.fits')
```

Note

The members of the WCS object correspond roughly to the key/value pairs in the FITS header. However, they are adjusted and normalized in a number of ways that make performing the WCS transformation easier. Therefore, they can not be relied upon to get the original values in the header. To build up a FITS header directly and specifically, use `astropy.io.fits.Header` directly.

Second Example

Another way of creating a WCS object is via the use of a Python dictionary. This affords us more control over the `NAXISn` FITS header keyword which is otherwise automatically default to zero as in the case of the First Example shown above.

```
# Define the astropy.wcs.WCS object using a Python dictionary as
input

import astropy.wcs
wcs_dict = {
'CTYPE1': 'WAVE    ', 'CUNIT1': 'Angstrom', 'CDELT1': 0.2, 'CRPIX1':
0, 'CRVAL1': 10, 'NAXIS1': 5,
'CTYPE2': 'HPLT-TAN', 'CUNIT2': 'deg', 'CDELT2': 0.5, 'CRPIX2': 2,
'CRVAL2': 0.5, 'NAXIS2': 4,
'CTYPE3': 'HPLN-TAN', 'CUNIT3': 'deg', 'CDELT3': 0.4, 'CRPIX3': 2,
'CRVAL3': 1, 'NAXIS3': 3}
input_wcs = astropy.wcs.WCS(wcs_dict)
```

Loading WCS Information from a FITS File

This example loads a FITS file (supplied on the command line) and uses the FITS keywords in its primary header to create a WCS and transform.

```
# Load the WCS information from a fits header, and use it
# to convert pixel coordinates to world coordinates.

import numpy as np
from astropy import wcs
from astropy.io import fits
import sys
```

```
def load_wcs_from_file(filename):
    # Load the FITS hdulist using astropy.io.fits
    hdulist = fits.open(filename)

    # Parse the WCS keywords in the primary HDU
    w = wcs.WCS(hdulist[0].header)

    # Print out the "name" of the WCS, as defined in the FITS header
    print(w.wcs.name)

    # Print out all of the settings that were parsed from the header
    w.wcs.print_contents()

    # Three pixel coordinates of interest.
    # Note we've silently assumed an NAXIS=2 image here.
    # The pixel coordinates are pairs of [X, Y].
    # The "origin" argument indicates whether the input coordinates
    # are 0-based (as in Numpy arrays) or
    # 1-based (as in the FITS convention, for example coordinates
    # coming from DS9).
    pixcrd = np.array([[0, 0], [24, 38], [45, 98]], dtype=np.float64)

    # Convert pixel coordinates to world coordinates
    # The second argument is "origin" -- in this case we're declaring
we
    # have 0-based (Numpy-like) coordinates.
    world = w.wcs_pix2world(pixcrd, 0)
    print(world)

    # Convert the same coordinates back to pixel coordinates.
    pixcrd2 = w.wcs_world2pix(world, 0)
    print(pixcrd2)

    # These should be the same as the original pixel coordinates,
modulo
    # some floating-point error.
    assert np.max(np.abs(pixcrd - pixcrd2)) < 1e-6

    # The example below illustrates the use of "origin" to convert
between
    # 0- and 1- based coordinates when executing the forward and
backward
    # WCS transform.
    x = 0
    y = 0
    origin = 0
    assert (w.wcs_pix2world(x, y, origin) ==
            w.wcs_pix2world(x + 1, y + 1, origin + 1))


if __name__ == '__main__':
    load_wcs_from_file(sys.argv[-1])
```

WCS Tools

Subsetting and Pixel Scales

WCS objects can be broken apart into their constituent axes using the **`sub`** function. There is also a **`celestial`** convenience function that will return a WCS object with only the celestial axes included.

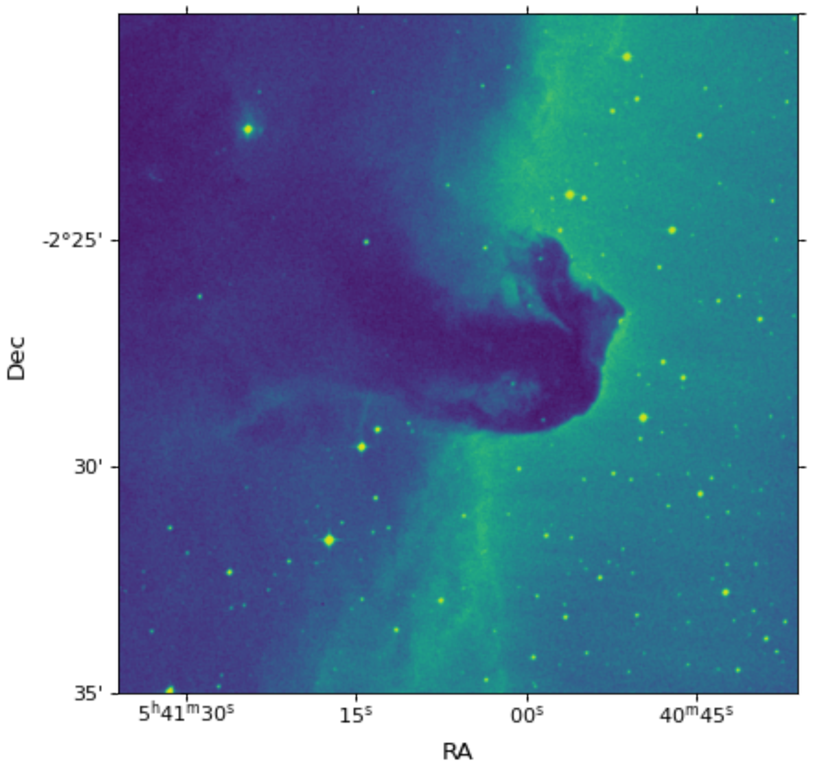
The pixel scales of a celestial image or the pixel dimensions of a non-celestial image can be extracted with the utility functions **`proj_plane_pixel_scales`** and **`non_celestial_pixel_scales`**. Likewise, celestial pixel area can be extracted with the utility function **`proj_plane_pixel_area`**.

Matplotlib plots with correct WCS projection

The WCSAxes framework, previously a standalone package, allows the **`WCS`** to be used to define projections in Matplotlib. More information on using WCSAxes can be found here.

```
from matplotlib import pyplot as plt
from astropy.io import fits
from astropy.wcs import WCS
from astropy.utils.data import get_pkg_data_filename

filename = get_pkg_data_filename('tutorials/FITS-images
/HorseHead.fits')

hdu = fits.open(filename)[0]
wcs = WCS(hdu.header)

fig = plt.figure()
fig.add_subplot(111, projection=wcs)
plt.imshow(hdu.data, origin='lower', cmap=plt.cm.viridis)
plt.xlabel('RA')
plt.ylabel('Dec')
```

(png, svg, pdf)



Relax Constants

The `relax` keyword argument controls the handling of non-standard FITS WCS keywords.

Note that the default value of `relax` is **`True`** for reading (to accept all non standard keywords), and **`False`** for writing (to write out only standard keywords), in accordance with Postel's prescription:

> "Be liberal in what you accept, and conservative in what you send."

Header-reading relaxation constants

`WCS`, **`Wcsprm`** and **`find_all_wcs`** have a *relax* argument, which may be either **`True`**, **`False`** or an **`int`**.

- If **`True`**, (default), all non-standard WCS extensions recognized by the parser will be handled.
- If **`False`**, none of the extensions (even those in the errata) will be handled. Non-conformant keywords will be handled in the same way as non-WCS keywords in the header, i.e. by simply ignoring them.
- If an **`int`**, is is a bit field to provide fine-grained control over what non-standard WCS keywords to accept. The flag bits are subject to change in future and should be set by using the constants beginning with `WCSHDR_` in the **`astropy.wcs`** module.

For example, to accept `CD00i00j` and `PC00i00j` use:

```
relax = astropy.wcs.WCSHDR_CD00i00j | astropy.wcs.WCSHDR_PC00i00j
```

The parser always treats `EPOCH` as subordinate to `EQUINOXa` if both are present, and `VSOURCEa` is always subordinate to `ZSOURCEa`.

Likewise, `VELREF` is subordinate to the formalism of WCS Paper III.

The flag bits are:

- `WCSHDR_none`: Don't accept any extensions (not even those in the errata). Treat non-conformant keywords in the same way as non-WCS keywords in the header, i.e. simply ignore them. (This is equivalent to passing **`False`**)
- `WCSHDR_all`: Accept all extensions recognized by the parser. (This is equivalent to the default behavior or passing **`True`**).
- `WCSHDR_reject`: Reject non-standard keyrecords (that are not otherwise explicitly accepted by one of the flags below). A warning will be displayed by default.

  This flag may be used to signal the presence of non-standard keywords, otherwise they are simply passed over as though they did not exist in the header. It is mainly intended for testing conformance of a FITS header to the WCS standard.

  Keyrecords may be non-standard in several ways:

  - The keyword may be syntactically valid but with keyvalue of incorrect type or invalid syntax, or the keycomment may be malformed.
  - The keyword may strongly resemble a WCS keyword but not, in fact, be one because it does not conform to the standard. For example, `CRPIX01` looks like a `CRPIXja` keyword, but in fact the leading zero on the axis number violates the basic FITS standard. Likewise, `LONPOLE2` is not a valid `LONPOLEa` keyword in the WCS standard, and indeed there is nothing the parser can sensibly do with it.
  - Use of the keyword may be deprecated by the standard. Such will be rejected if not explicitly accepted via one of the flags below.
- `WCSHDR_CROTAia`: Accept `CROTAia`, `iCROTna`, `TCROTna`
- `WCSHDR_EPOCHa`: Accept `EPOCHa`.
- `WCSHDR_VELREFa`: Accept `VELREFa`.

  The constructor always recognizes the AIPS-convention keywords, `CROTAn`, `EPOCH`, and `VELREF` for the primary representation `(a = ' ')` but alternates are non-standard.

The constructor accepts `EPOCHa` and `VELREFa` only if `WCSHDR_AUXIMG` is also enabled.

- `WCSHDR_CD00i00j` : Accept `CD00i00j` .
- `WCSHDR_PC00i00j` : Accept `PC00i00j` .
- `WCSHDR_PROJPn` : Accept `PROJPn` .

  These appeared in early drafts of WCS Paper I+II (before they were split) and are equivalent to `CDi_ja` , `PCi_ja` , and `PVi_ma` for the primary representation `(a = ' ')` . `PROJPn` is equivalent to `PVi_ma` with `m` = `n` <= 9, and is associated exclusively with the latitude axis.

- `WCSHDR_CD0i_0ja` : Accept `CD0i_0ja` (wcspih()).
- `WCSHDR_PC0i_0ja` : Accept `PC0i_0ja` (wcspih()).
- `WCSHDR_PV0i_0ma` : Accept `PV0i_0ja` (wcspih()).
- `WCSHDR_PS0i_0ma` : Accept `PS0i_0ja` (wcspih()).

  Allow the numerical index to have a leading zero in doubly-parameterized keywords, for example, `PC01_01` . WCS Paper I (Sects 2.1.2 & 2.1.4) explicitly disallows leading zeroes. The FITS 3.0 standard document (Sect. 4.1.2.1) states that the index in singly-parameterized keywords (e.g. `CTYPEia`) “shall not have leading zeroes”, and later in Sect. 8.1 that “leading zeroes must not be used” on `PVi_ma` and `PSi_ma` . However, by an oversight, it is silent on `PCi_ja` and `CDi_ja` .

  Only available if built with wcslib 5.0 or later.

- `WCSHDR_RADECSYS` : Accept `RADECSYS` . This appeared in early drafts of WCS Paper I+II and was subsequently replaced by `RADESYSa` . The constructor accepts `RADECSYS` only if `WCSHDR_AUXIMG` is also enabled.
- `WCSHDR_VSOURCE` : Accept `VSOURCEa` or `VSOUna` . This appeared in early drafts of WCS Paper III and was subsequently dropped in favor of `ZSOURCEa` and `ZSOUna` . The constructor accepts `VSOURCEa` only if `WCSHDR_AUXIMG` is also enabled.
- `WCSHDR_DOBSn` : Allow `DOBSn` , the column-specific analogue of `DATE-OBS` . By an oversight this was never formally defined in the standard.
- `WCSHDR_LONGKEY` : Accept long forms of the alternate binary table and pixel list WCS keywords, i.e. with “a” non- blank. Specifically:

```
jCRPXna  TCRPXna  :  jCRPXn  jCRPna  TCRPXn  TCRPna  CRPIXja
   -     TPCn_ka  :    -     ijPCna    -     TPn_ka  PCi_ja
   -     TCDn_ka  :    -     ijCDna    -     TCn_ka  CDi_ja
iCDLTna  TCDLTna  :  iCDLTn  iCDEna  TCDLTn  TCDEna  CDELTia
iCUNIna  TCUNIna  :  iCUNIn  iCUNna  TCUNIn  TCUNna  CUNITia
iCTYPna  TCTYPna  :  iCTYPn  iCTYna  TCTYPn  TCTYna  CTYPEia
iCRVLna  TCRVLna  :  iCRVLn  iCRVna  TCRVLn  TCRVna  CRVALia
iPVn_ma  TPVn_ma  :    -     iVn_ma    -     TVn_ma  PVi_ma
iPSn_ma  TPSn_ma  :    -     iSn_ma    -     TSn_ma  PSi_ma
```

where the primary and standard alternate forms together with the image-header equivalent are shown rightwards of the colon.

The long form of these keywords could be described as quasi- standard. `TPCn_ka`, `iPVn_ma`, and `TPVn_ma` appeared by mistake in the examples in WCS Paper II and subsequently these and also `TCDn_ka`, `iPSn_ma` and `TPSn_ma` were legitimized by the errata to the WCS papers.

Strictly speaking, the other long forms are non-standard and in fact have never appeared in any draft of the WCS papers nor in the errata. However, as natural extensions of the primary form they are unlikely to be written with any other intention. Thus it should be safe to accept them provided, of course, that the resulting keyword does not exceed the 8-character limit.

If `WCSHDR_CNAMn` is enabled then also accept:

```
iCNAMna  TCNAMna  :   ---   iCNAna    ---   TCNAna  CNAMEia
iCRDEna  TCRDEna  :   ---   iCRDna    ---   TCRDna  CRDERia
iCSYEna  TCSYEna  :   ---   iCSYna    ---   TCSYna  CSYERia
```

Note that `CNAMEia`, `CRDERia`, `CSYERia`, and their variants are not used by **`astropy.wcs`** but are stored as auxiliary information.

- `WCSHDR_CNAMn`: Accept `iCNAMn`, `iCRDEn`, `iCSYEn`, `TCNAMn`, `TCRDEn`, and `TCSYEn`, i.e. with `a` blank. While non-standard, these are the analogues of `iCTYPn`, `TCTYPn`, etc.
- `WCSHDR_AUXIMG`: Allow the image-header form of an auxiliary WCS keyword with representation-wide scope to provide a default value for all images. This default may be overridden by the column-specific form of the keyword.

  For example, a keyword like `EQUINOXa` would apply to all image arrays in a binary table, or all pixel list columns with alternate representation `a` unless overridden by `EQUIna`.

Specifically the keywords are:

```
LATPOLEa  for LATPna
LONPOLEa  for LONPna
RESTFREQ  for RFRQna
RESTFRQa  for RFRQna
RESTWAVa  for RWAVna
```

whose keyvalues are actually used by WCSLIB, and also keywords that provide auxiliary information that is simply stored in the wcsprm struct:

```
EPOCH         -       ... (No column-specific form.)
EPOCHa        -       ... Only if WCSHDR_EPOCHa is set.
EQUINOXa  for EQUIna
RADESYSa  for RADEna
RADECSYS  for RADEna  ... Only if WCSHDR_RADECSYS is set.
SPECSYSa  for SPECna
SSYSOBSa  for SOBSna
SSYSSRCa  for SSRCna
VELOSYSa  for VSYSna
VELANGLa  for VANGna
VELREF        -       ... (No column-specific form.)
VELREFa       -       ... Only if WCSHDR_VELREFa is set.
VSOURCEa  for VSOUna  ... Only if WCSHDR_VSOURCE is set.
WCSNAMEa  for WCSNna  ... Or TWCSna (see below).
ZSOURCEa  for ZSOUna

DATE-AVG  for DAVGn
DATE-OBS  for DOBSn
MJD-AVG   for MJDAn
MJD-OBS   for MJDOBn
OBSGEO-X  for OBSGXn
OBSGEO-Y  for OBSGYn
OBSGEO-Z  for OBSGZn
```

where the image-header keywords on the left provide default values for the column specific keywords on the right.

Keywords in the last group, such as `MJD-OBS`, apply to all alternate representations, so `MJD-OBS` would provide a default value for all images in the header.

This auxiliary inheritance mechanism applies to binary table image arrays and pixel lists alike. Most of these keywords have no default value, the exceptions being `LONPOLEa` and `LATPOLEa`, and also `RADESYSa` and `EQUINOXa` which provide defaults for each other. Thus the only potential difficulty in using `WCSHDR_AUXIMG` is that of erroneously inheriting one of these four keywords.

Unlike `WCSHDR_ALLIMG`, the existence of one (or all) of these auxiliary WCS image header keywords will not by itself cause a **Wcsprm** object to be created for alternate representation `a`. This is because they do not provide sufficient information to create a non-trivial coordinate representation when used in conjunction with the default values of those keywords, such as `CTYPEia`, that are parameterized by axis number.

- `WCSHDR_ALLIMG`: Allow the image-header form of *all* image header WCS keywords to provide a default value for all image arrays in a binary table (n.b. not pixel list). This default may be overridden by the column-specific form of the keyword.

  For example, a keyword like `CRPIXja` would apply to all image arrays in a binary table with alternate representation `a` unless overridden by `jCRPna`.

  Specifically the keywords are those listed above for `WCSHDR_AUXIMG` plus:

```
WCSAXESa  for WCAXna
```

  which defines the coordinate dimensionality, and the following keywords which are parameterized by axis number:

```
CRPIXja   for jCRPna
PCi_ja    for ijPCna
CDi_ja    for ijCDna
CDELTia   for iCDEna
CROTAi    for iCROTn
CROTAia      -      ... Only if WCSHDR_CROTAia is set.
CUNITia   for iCUNna
CTYPEia   for iCTYna
CRVALia   for iCRVna
PVi_ma    for iVn_ma
PSi_ma    for iSn_ma

CNAMEia   for iCNAna
CRDERia   for iCRDna
CSYERia   for iCSYna
```

  where the image-header keywords on the left provide default values for the column specific keywords on the right.

  This full inheritance mechanism only applies to binary table image arrays, not pixel lists, because in the latter case there is no well-defined association between coordinate axis number and column number.

  Note that `CNAMEia`, `CRDERia`, `CSYERia`, and their variants are not used by pywcs but are stored in the `Wcsprm` object as auxiliary information.

Note especially that at least one `Wcsprm` object will be returned for each `a` found in one of the image header keywords listed above:

- If the image header keywords for `a` **are not** inherited by a binary table, then the struct will not be associated with any particular table column number and it is up to the user to provide an association.
- If the image header keywords for `a` **are** inherited by a binary table image array, then those keywords are considered to be "exhausted" and do not result in a separate `Wcsprm` object.

Header-writing relaxation constants

`to_header` and `to_header_string` has a *relax* argument which may be either `True`, `False` or an `int`.

- If `True`, write all recognized extensions.
- If `False` (default), write all extensions that are considered to be safe and recommended, equivalent to `WCSHDO_safe` (described below).
- If an `int`, is is a bit field to provide fine-grained control over what non-standard WCS keywords to accept. The flag bits are subject to change in future and should be set by using the constants beginning with `WCSHDO_` in the `astropy.wcs` module.

The flag bits are:

- `WCSHDO_none` : Don't use any extensions.
- `WCSHDO_all` : Write all recognized extensions, equivalent to setting each flag bit.
- `WCSHDO_safe` : Write all extensions that are considered to be safe and recommended.
- `WCSHDO_DOBSn` : Write `DOBSn`, the column-specific analogue of `DATE-OBS` for use in binary tables and pixel lists. WCS Paper III introduced `DATE-AVG` and `DAVGn` but by an oversight `DOBSn` was never formally defined by the standard. The alternative to using `DOBSn` is to write `DATE-OBS` which applies to the whole table. This usage is considered to be safe and is recommended.
- `WCSHDO_TPCn_ka` : WCS Paper I defined
  - `TPn_ka` and `TCn_ka` for pixel lists

  but WCS Paper II uses `TPCn_ka` in one example and subsequently the errata for the WCS papers legitimized the use of

- `TPCn_ka` and `TCDn_ka` for pixel lists

provided that the keyword does not exceed eight characters. This usage is considered to be safe and is recommended because of the non-mnemonic terseness of the shorter forms.

- `WCSHDO_PVn_ma`: WCS Paper I defined
  - `iVn_ma` and `iSn_ma` for bintables and
  - `TVn_ma` and `TSn_ma` for pixel lists

  but WCS Paper II uses `iPVn_ma` and `TPVn_ma` in the examples and subsequently the errata for the WCS papers legitimized the use of

  - `iPVn_ma` and `iPSn_ma` for bintables and
  - `TPVn_ma` and `TPSn_ma` for pixel lists

  provided that the keyword does not exceed eight characters. This usage is considered to be safe and is recommended because of the non-mnemonic terseness of the shorter forms.
- `WCSHDO_CRPXna`: For historical reasons WCS Paper I defined
  - `jCRPXn`, `iCDLTn`, `iCUNIn`, `iCTYPn`, and `iCRVLn` for bintables and
  - `TCRPXn`, `TCDLTn`, `TCUNIn`, `TCTYPn`, and `TCRVLn` for pixel lists

  for use without an alternate version specifier. However, because of the eight-character keyword constraint, in order to accommodate column numbers greater than 99 WCS Paper I also defined

  - `jCRPna`, `iCDEna`, `iCUNna`, `iCTYna` and `iCRVna` for bintables and
  - `TCRPna`, `TCDEna`, `TCUNna`, `TCTYna` and `TCRVna` for pixel lists

  for use with an alternate version specifier (the `a`). Like the `PC`, `CD`, `PV`, and `PS` keywords there is a tendency to confuse these two forms for column numbers up to 99. It is very unlikely that any parser would reject keywords in the first set with a non-blank alternate version specifier so this usage is considered to be safe and is recommended.
- `WCSHDO_CNAMna`: WCS Papers I and III defined
  - `iCNAna`, `iCRDna`, and `iCSYna` for bintables and
  - `TCNAna`, `TCRDna`, and `TCSYna` for pixel lists

  By analogy with the above, the long forms would be

  - `iCNAMna`, `iCRDEna`, and `iCSYEna` for bintables and

  - `TCNAMna`, `TCRDEna`, and `TCSYEna` for pixel lists

    Note that these keywords provide auxiliary information only, none of them are needed to compute world coordinates. This usage is potentially unsafe and is not recommended at this time.

- `WCSHDO_WCSNna`: Write `WCSNna` instead of `TWCSna` for pixel lists. While the constructor treats `WCSNna` and `TWCSna` as equivalent, other parsers may not. Consequently, this usage is potentially unsafe and is not recommended at this time.
- `WCSHDO_SIP`: Write out Simple Imaging Polynomial (SIP) keywords.
- `WCSHDO_P12`, `WCSHDO_P13`, `WCSHDO_P14`, `WCSHDO_P15`, `WCSHDO_P16`, `WCSHDO_P17`, `WCSHDO_EFMT`

  These constants control the precision of the WCS keywords returned by **`to_header`**.

  - `WCSHDO_P12` : Use "%20.12G" format for all floating-point keyvalues (12 significant digits)
  - `WCSHDO_P13` : Use "%21.13G" format for all floating-point keyvalues (13 significant digits)
  - `WCSHDO_P14` : Use "%22.14G" format for all floating-point keyvalues (14 significant digits)
  - `WCSHDO_P15` : Use "%23.15G" format for all floating-point keyvalues (15 significant digits)
  - `WCSHDO_P16` : Use "%24.16G" format for all floating-point keyvalues (16 significant digits)
  - `WCSHDO_P17` : Use "%25.17G" format for all floating-point keyvalues (17 significant digits)
  - `WCSHDO_EFMT` : Use "%E" format instead of the default "%G" format above

Other Information

astropy.wcs History

`astropy.wcs` began life as `pywcs`. Earlier version numbers refer to that package.

pywcs Version 1.11

- Updated to wcslib version 4.8, which gives much more detailed error messages.

- Added functions get_pc() and get_cdelt(). These provide a way to always get the canonical representation of the linear transformation matrix, whether the header specified it in PC, CD or CROTA form.
- Long-running process will now release the Python GIL to better support Python multithreading.
- The dimensions of the `cd` and `pc` matrices were always returned as 2x2. They now are sized according to naxis.
- Supports Python 3.x
- Builds on Microsoft Windows without severely patching wcslib.
- Lots of new unit tests
- `pywcs` will now run without `pyfits`, though the SIP and distortion lookup table functionality is unavailable.
- Setting `cunit` will now verify that the values are valid unit strings.

pywcs Version 1.10

- Adds a `UnitConversion` class, which gives access to wcslib's unit conversion functionality. Given two convertible unit strings, pywcs can convert arrays of values from one to the other.
- Now uses wcslib 4.7
- Changes to some wcs values would not always calculate secondary values.

pywcs Version 1.9

- Support binary image arrays and pixel list format WCS by presenting a way to call wcslib's `wcsbth()`
- Updated underlying wcslib to version 4.5, which fixes the following:
    - Fixed the interpretation of VELREF when translating AIPS-convention spectral types. Such translation is now handled by a new special- purpose function, spcaips(). The wcsprm struct has been augmented with an entry for velref which is filled by wcspih() and wcsbth(). Previously, selection by VELREF of the radio or optical velocity convention for type VELO was not properly handled.

Bugs

- The `pc` member is now available with a default raw `Wcsprm` object.
- Make properties that return arrays read-only, since modifying a (mutable)

array could result in secondary values not being recomputed based on those changes.

- `float` properties can now be set using `int` values

pywcs Version 1.3a1

Earlier versions of pywcs had two versions of every conversion method:

```
X(...)      -- treats the origin of pixel coordinates at (0, 0)
X_fits(...) -- treats the origin of pixel coordinates at (1, 1)
```

From version 1.3 onwards, there is only one method for each conversion, with an ‘origin’ argument:

- 0: places the origin at (0, 0), which is the C/Numpy convention.
- 1: places the origin at (1, 1), which is the Fortran/FITS convention.

Validation and Bounds checking

Bounds checking is enabled by default, and any computed world coordinates outside of [-180°, 180°] for longitude and [-90°, 90°] in latitude are marked as invalid. To disable this behavior, use `astropy.wcs.Wcsprm.bounds_check`.

Reference/API

Reference/API

astropy.wcs Package

`astropy.wcs` contains utilities for managing World Coordinate System (WCS) transformations in FITS files. These transformations map the pixel locations in an image to their real-world units, such as their position on the sky sphere.

It performs three separate classes of WCS transformations:

- Core WCS, as defined in the FITS WCS standard, based on Mark Calabretta’s wcslib. See `Wcsprm`.
- Simple Imaging Polynomial (SIP) convention. See `Sip`.
- table lookup distortions as defined in WCS distortion paper. See `DistortionLookupTable`.

Each of these transformations can be used independently or together in a standard pipeline.

Functions

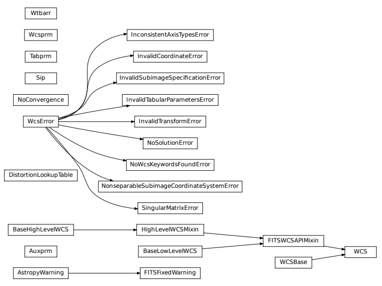
`find_all_wcs`(header[, relax, keysel, fix, …])	Find all the WCS transformations in the given header.
`get_include`()	Get the path to astropy.wcs's C header files.
`validate`(source)	Prints a WCS validation report for the given FITS file.

Classes

`Auxprm`()	Class that contains auxiliary coordinate system information of a specialist nature.
`DistortionLookupTable`(*table*, *crpix*, …)	Represents a single lookup table for a distortion paper transformation.
`FITSFixedWarning`	The warning raised when the contents of the FITS header have been modified to be standards compliant.
`InconsistentAxisTypesError`()	The WCS header inconsistent or unrecognized coordinate axis type(s).
`InvalidCoordinateError`()	One or more of the world coordinates is invalid.
`InvalidSubimageSpecificationError`()	The subimage specification is invalid.
`InvalidTabularParametersError`()	The given tabular parameters are invalid.
`InvalidTransformError`()	The WCS transformation is invalid, or the transformation parameters are invalid.
`NoConvergence`(*args[, best_solution, …])	An error class used to report non-convergence and/or divergence of numerical methods.
`NoSolutionError`()	No solution can be found in the given interval.
`NoWcsKeywordsFoundError`()	No WCS keywords were found in the given header.
`NonseparableSubimageCoordinateSystemError`()	Non-separable subimage coordinate system.
`SingularMatrixError`()	The linear transformation matrix is singular.
`Sip`(*a, b, ap, bp, crpix*)	The `Sip` class performs polynomial distortion correction using the SIP convention in both directions.
`Tabprm`()	A class to store the information related to tabular coordinates, i.e., coordinates that are defined via a lookup table.
`WCS`([header, fobj, key, minerr, relax, …])	WCS objects perform standard WCS transformations, and correct for SIP and distortion paper table-lookup transformations, based on the WCS keywords and supplementary data read from a FITS file.
`WCSBase`(*sip, cpdis, wcsprm, det2im*)	Wcs objects amalgamate basic WCS (as provided by wcslib), with SIP and distortion paper operations.
`WcsError`	Base class of all invalid WCS errors.

Wcsprm([header, key, relax, naxis, keysel, …])	**Wcsprm** performs the core WCS transformations.
Wtbarr()	Classes to construct coordinate lookup tables from a binary table extension (BINTABLE).

Class Inheritance Diagram



astropy.wcs.utils Module

Functions

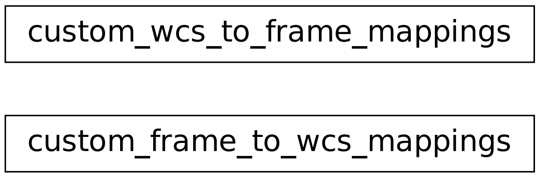
add_stokes_axis_to_wcs(wcs, add_before_ind)	Add a new Stokes axis that is uncorrelated with any other axes.
celestial_frame_to_wcs(frame[, projection])	For a given coordinate frame, return the corresponding WCS object.
wcs_to_celestial_frame(wcs)	For a given WCS, return the coordinate frame that matches the celestial component of the WCS.
proj_plane_pixel_scales(wcs)	For a WCS returns pixel scales along each axis of the image pixel at the `CRPIX` location once it is projected onto the “plane of intermediate world coordinates” as defined in Greisen & Calabretta 2002, A&A, 395, 1061.
proj_plane_pixel_area(wcs)	For a **celestial** WCS (see **astropy.wcs.WCS.celestial**) returns pixel area of the image pixel at the `CRPIX` location once it is projected onto the “plane of intermediate world coordinates” as defined in Greisen & Calabretta 2002, A&A, 395, 1061.
is_proj_plane_distorted(wcs[, maxerr])	For a WCS returns **False** if square image (detector) pixels stay square when projected onto the “plane of intermediate world coordinates” as defined in Greisen & Calabretta 2002, A&A, 395, 1061.
non_celestial_pixel_scales(inwcs)	Calculate the pixel scale along each axis of a non-celestial WCS, for example one with mixed spectral and spatial axes.
skycoord_to_pixel(coords, wcs[, origin, mode])	Convert a set of SkyCoord coordinates into pixels.
pixel_to_skycoord(xp, yp, wcs[, origin, …])	Convert a set of pixel coordinates into a **SkyCoord** coordinate.

`pixel_to_pixel`(wcs_in, wcs_out, *inputs)	Transform pixel coordinates in a dataset with a WCS to pixel coordinates in another dataset with a different WCS.
`local_partial_pixel_derivatives`(wcs, *pixel)	Return a matrix of shape `(world_n_dim, pixel_n_dim)` where each entry `[i, j]` is the partial derivative d(world_i)/d(pixel_j) at the requested pixel position.
`fit_wcs_from_points`(xy, world_coords[, …])	Given two matching sets of coordinates on detector and sky, compute the WCS.

Classes

`custom_wcs_to_frame_mappings`([mappings])	
`custom_frame_to_wcs_mappings`([mappings])	

Class Inheritance Diagram



astropy.wcs.wcsapi Package

Functions

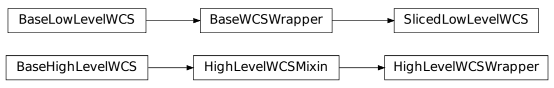
`deserialize_class`(tpl[, construct])	Deserialize classes recursively.
`sanitize_slices`(slices, ndim)	Given a slice as input sanitise it to an easier to parse format.format
`validate_physical_types`(physical_types)	Validate a list of physical types against the UCD1+ standard
`wcs_info_str`(wcs)	

Classes

`BaseHighLevelWCS`()	Abstract base class for the high-level WCS interface.
`BaseLowLevelWCS`()	Abstract base class for the low-level WCS interface.
`BaseWCSWrapper`(wcs, *args, **kwargs)	A base wrapper class for things that modify Low Level WCSes.
`HighLevelWCSMixin`()	Mix-in class that automatically provides the high-level WCS API for the low-level WCS object given by the `low_level_wcs` property.

HighLevelWCSWrapper(low_level_wcs)	Wrapper class that can take any **BaseLowLevelWCS** object and expose the high-level WCS API.
SlicedLowLevelWCS(wcs, slices)	A Low Level WCS wrapper which applies an array slice to a WCS.

Class Inheritance Diagram



See Also

- wcslib

Acknowledgments and Licenses

wcslib is licenced under the GNU Lesser General Public License.

Models and Fitting (`astropy.modeling`)

Introduction

`astropy.modeling` provides a framework for representing models and performing model evaluation and fitting. A number of predefined 1-D and 2-D models are provided and the capability for custom, user defined models is supported. Different fitting algorithms can be used with any model. For those fitters with the capabilities fitting can be done using uncertainties, parameters with bounds, and priors.

Note

A number of significant changes have been made to the internals that have been documented in more detail in Changes to Modeling in v4.0. The main change is that combining model classes no longer is supported. (Combining model instances is still very much supported!)

Using Modeling

MODELS

Basics

The **astropy.modeling** package defines a number of models that are collected under a single namespace as `astropy.modeling.models`. Models behave like parametrized functions:

```
>>> import numpy as np
>>> from astropy.modeling import models
>>> g = models.Gaussian1D(amplitude=1.2, mean=0.9, stddev=0.5)
>>> print(g)
Model: Gaussian1D
Inputs: ('x',)
Outputs: ('y',)
Model set size: 1
Parameters:
    amplitude mean stddev
    --------- ---- ------
          1.2  0.9    0.5
```

Model parameters can be accessed as attributes:

```
>>> g.amplitude
Parameter('amplitude', value=1.2)
>>> g.mean
Parameter('mean', value=0.9)
>>> g.stddev
Parameter('stddev', value=0.5, bounds=(1.1754943508222875e-38, None))
```

and can also be updated via those attributes:

```
>>> g.amplitude = 0.8
>>> g.amplitude
Parameter('amplitude', value=0.8)
```

Models can be evaluated by calling them as functions:

```
>>> g(0.1)
0.22242984036255528
>>> g(np.linspace(0.5, 1.5, 7))
array([0.58091923, 0.71746405, 0.7929204 , 0.78415894, 0.69394278,
       0.54952605, 0.3894018 ])
```

As the above example demonstrates, in general most models evaluate array-like inputs according to the standard Numpy broadcasting rules for arrays. Models can therefore already be useful to evaluate common functions, independently of the fitting features of the package.

Instantiating and Evaluating Models

In general, models are instantiated by supplying the parameter values that define that instance of the model to the constructor, as demonstrated in the section on Parameters.

Additionally, a **`Model`** instance may represent a single model with one set of parameters, or a Model set consisting of a set of parameters each representing a different parameterization of the same parametric model. For example, you may instantiate a single Gaussian model with one mean, standard deviation, and amplitude. Or you may create a set of N Gaussians, each one of which would be evaluated on, for example, a different plane in an image cube.

For example, a single Gaussian model may be instantiated with all scalar parameters:

```
>>> from astropy.modeling.models import Gaussian1D
>>> g = Gaussian1D(amplitude=1, mean=0, stddev=1)
>>> g
<Gaussian1D(amplitude=1., mean=0., stddev=1.)>
```

The newly created model instance `g` now works like a Gaussian function with the specific parameters. It takes a single input:

```
>>> g.inputs
('x',)
>>> g(x=0)
1.0
```

The model can also be called without explicitly using keyword arguments:

```
>>> g(0)
1.0
```

Or a set of Gaussians may be instantiated by passing multiple parameter values:

```
>>> from astropy.modeling.models import Gaussian1D
>>> gset = Gaussian1D(amplitude=[1, 1.5, 2],
...                   mean=[0, 1, 2],
...                   stddev=[1., 1., 1.],
...                   n_models=3)
>>> print(gset)
Model: Gaussian1D
Inputs: ('x',)
Outputs: ('y',)
Model set size: 3
```

```
Parameters:
    amplitude mean stddev
    --------- ---- ------
          1.0  0.0    1.0
          1.5  1.0    1.0
          2.0  2.0    1.0
```

This model also works like a Gaussian function. The three models in the model set can be evaluated on the same input:

```
>>> gset(1.)
array([0.60653066, 1.5       , 1.21306132])
```

or on `N=3` inputs:

```
>>> gset([1, 2, 3])
array([0.60653066, 0.90979599, 1.21306132])
```

For a comprehensive example of fitting a model set see Fitting Model Sets.

Model inverses

All models have a **`Model.inverse`** property which may, for some models, return a new model that is the analytic inverse of the model it is attached to. For example:

```
>>> from astropy.modeling.models import Linear1D
>>> linear = Linear1D(slope=0.8, intercept=1.0)
>>> linear.inverse
<Linear1D(slope=1.25, intercept=-1.25)>
```

The inverse of a model will always be a fully instantiated model in its own right, and so can be evaluated directly like:

```
>>> linear.inverse(2.0)
1.25
```

It is also possible to assign a *custom* inverse to a model. This may be useful, for example, in cases where a model does not have an analytic inverse, but may have an approximate inverse that was computed numerically and is represented by another model. This works even if the target model has a default analytic inverse–in this case the default is overridden with the custom inverse:

```
>>> from astropy.modeling.models import Polynomial1D
>>> linear.inverse = Polynomial1D(degree=1, c0=-1.25, c1=1.25)
>>> linear.inverse
<Polynomial1D(1, c0=-1.25, c1=1.25)>
```

If a custom inverse has been assigned to a model, it can be deleted with `del model.inverse`. This resets the inverse to its default (if one exists). If a default does not exist, accessing `model.inverse` raises a **`NotImplementedError`**. For example polynomial models do not have a default inverse:

```
>>> del linear.inverse
>>> linear.inverse
<Linear1D(slope=1.25, intercept=-1.25)>
>>> p = Polynomial1D(degree=2, c0=1.0, c1=2.0, c2=3.0)
>>> p.inverse
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "astropy\modeling\core.py", line 796, in inverse
    raise NotImplementedError("An analytical inverse transform has
not "
NotImplementedError: No analytical or user-supplied inverse transform
has been implemented for this model.
```

One may certainly compute an inverse and assign it to a polynomial model though.

Note

When assigning a custom inverse to a model no validation is performed to ensure that it is actually an inverse or even approximate inverse. So assign custom inverses at your own risk.

Bounding Boxes

Efficient Model Rendering with Bounding Boxes

All **`Model`** subclasses have a **`bounding_box`** attribute that can be used to set the limits over which the model is significant. This greatly improves the efficiency of evaluation when the input range is much larger than the characteristic width of the model itself. For example, to create a sky model image from a large survey catalog, each source should only be evaluated over the pixels to which it contributes a significant amount of flux. This task can otherwise be computationally prohibitive on an average CPU.

The **`Model.render`** method can be used to evaluate a model on an output array, or input coordinate arrays, limiting the evaluation to the **`bounding_box`** region if it is set. This function will also produce postage stamp images of the model if no other input array is passed. To instead extract postage stamps from the data array itself, see 2D Cutout Images.

Using the Bounding Box

For basic usage, see `Model.bounding_box`. By default no `bounding_box` is set, except on model subclasses where a `bounding_box` property or method is explicitly defined. The default is then the minimum rectangular region symmetric about the position that fully contains the model. If the model does not have a finite extent, the containment criteria are noted in the documentation. For example, see `Gaussian2D.bounding_box`.

`Model.bounding_box` can be set by the user to any callable. This is particularly useful for models created with `custom_model` or as a `CompoundModel`:

```
>>> from astropy.modeling import custom_model
>>> def ellipsoid(x, y, z, x0=0, y0=0, z0=0, a=2, b=3, c=4, amp=1):
...     rsq = ((x - x0) / a) ** 2 + ((y - y0) / b) ** 2 + ((z - z0) /
c) ** 2
...     val = (rsq < 1) * amp
...     return val
...
>>> class Ellipsoid3D(custom_model(ellipsoid)):
...     # A 3D ellipsoid model
...     @property
...     def bounding_box(self):
...         return ((self.z0 - self.c, self.z0 + self.c),
...                 (self.y0 - self.b, self.y0 + self.b),
...                 (self.x0 - self.a, self.x0 + self.a))
...
>>> model = Ellipsoid3D()
>>> model.bounding_box
((-4.0, 4.0), (-3.0, 3.0), (-2.0, 2.0))
```

By default models are evaluated on any inputs. By passing a flag they can be evaluated only on inputs within the bounding box. For inputs outside of the bounding_box a `fill_value` is returned (`np.nan` by default):

```
>>> model(-5, 1, 1)
0.0
>>> model(-5, 1, 1, with_bounding_box=True)
nan
>>> model(-5, 1, 1, with_bounding_box=True, fill_value=-1)
-1.0
```

Warning

Currently when combining models the bounding boxes of components are combined only when joining models with the `&` operator. For the other operators bounding boxes for compound models must be assigned explicitly. A future release will determine the appropriate bounding box for a compound model where possible.

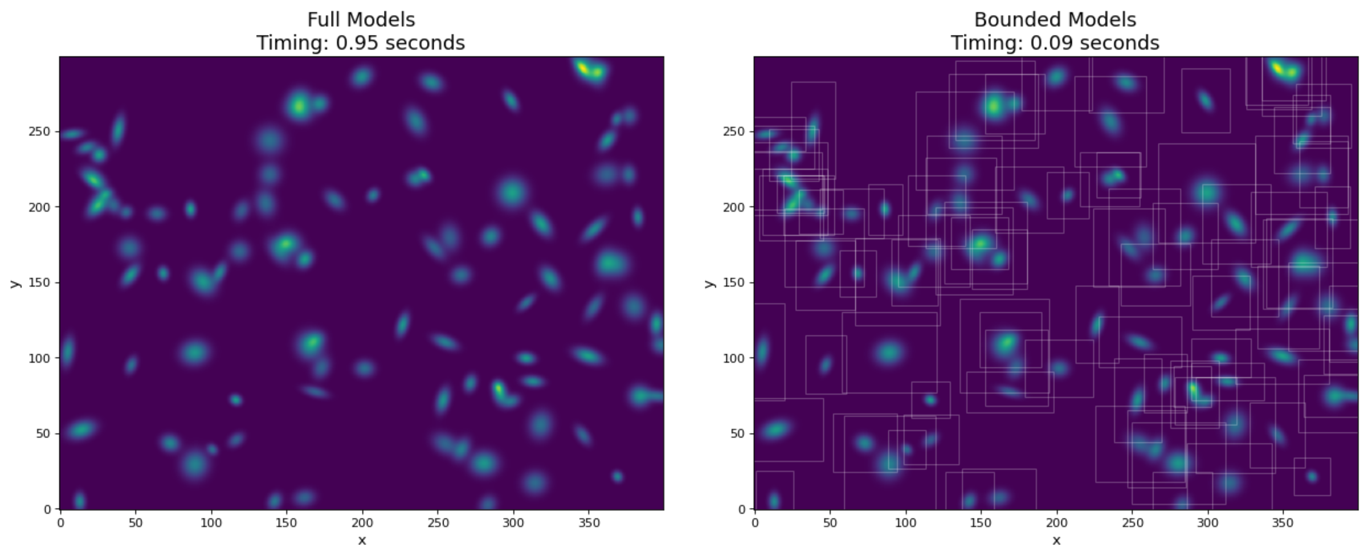
Efficient evaluation with `Model.render()`

When a model is evaluated over a range much larger than the model itself, it may be prudent to use the `Model.render` method if efficiency is a concern. The `render` method can be used to evaluate the model on an array of the same dimensions. `model.render()` can be called with no arguments to return a "postage stamp" of the bounding box region.

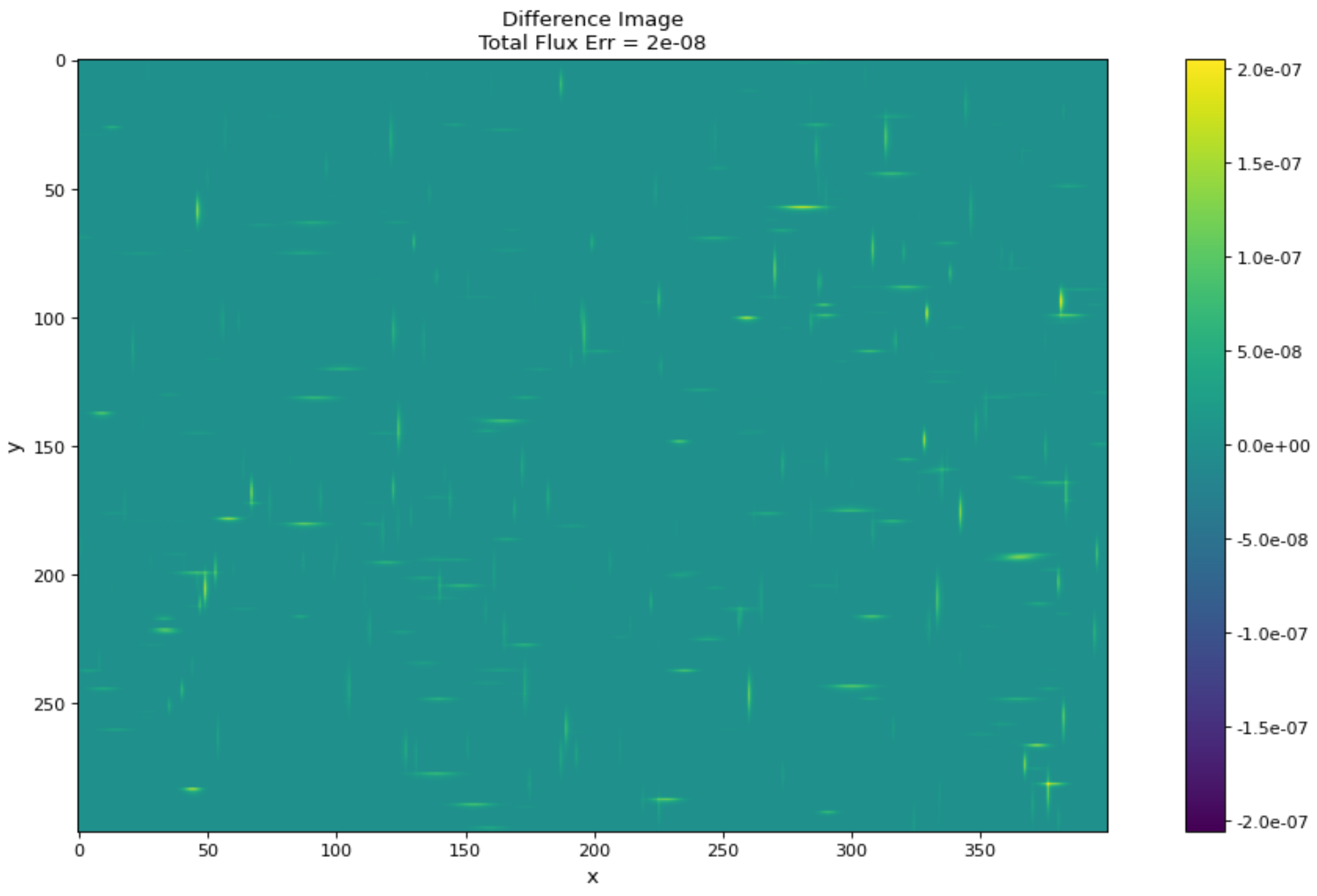
In this example, we generate a 300x400 pixel image of 100 2D Gaussian sources. For comparison, the models are evaluated both with and without using bounding boxes. By using bounding boxes, the evaluation speed increases by approximately a factor of 10 with negligible loss of information.

```
import numpy as np
from time import time
from astropy.modeling import models
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle

imshape = (300, 400)
y, x = np.indices(imshape)

# Generate random source model list
np.random.seed(0)
nsrc = 100
model_params = [
    dict(amplitude=np.random.uniform(.5, 1),
         x_mean=np.random.uniform(0, imshape[1] - 1),
         y_mean=np.random.uniform(0, imshape[0] - 1),
         x_stddev=np.random.uniform(2, 6),
         y_stddev=np.random.uniform(2, 6),
         theta=np.random.uniform(0, 2 * np.pi))
    for _ in range(nsrc)]

model_list = [models.Gaussian2D(**kwargs) for kwargs in model_params]

# Render models to image using bounding boxes
bb_image = np.zeros(imshape)
t_bb = time()
for model in model_list:
    model.render(bb_image)
t_bb = time() - t_bb

# Render models to image using full evaluation
```

```
full_image = np.zeros(imshape)
t_full = time()
for model in model_list:
    model.bounding_box = None
    model.render(full_image)
t_full = time() - t_full

flux = full_image.sum()
diff = (full_image - bb_image)
max_err = diff.max()

# Plots
plt.figure(figsize=(16, 7))
plt.subplots_adjust(left=.05, right=.97, bottom=.03, top=.97,
wspace=0.15)

# Full model image
plt.subplot(121)
plt.imshow(full_image, origin='lower')
plt.title('Full Models\nTiming: {:.2f} seconds'.format(t_full),
fontsize=16)
plt.xlabel('x')
plt.ylabel('y')

# Bounded model image with boxes overplotted
ax = plt.subplot(122)
plt.imshow(bb_image, origin='lower')
for model in model_list:
    del model.bounding_box  # Reset bounding_box to its default
    dy, dx = np.diff(model.bounding_box).flatten()
    pos = (model.x_mean.value - dx / 2, model.y_mean.value - dy / 2)
    r = Rectangle(pos, dx, dy, edgecolor='w', facecolor='none',
alpha=.25)
    ax.add_patch(r)
plt.title('Bounded Models\nTiming: {:.2f} seconds'.format(t_bb),
fontsize=16)
plt.xlabel('x')
plt.ylabel('y')

# Difference image
plt.figure(figsize=(16, 8))
plt.subplot(111)
plt.imshow(diff, vmin=-max_err, vmax=max_err)
plt.colorbar(format='%.1e')
plt.title('Difference Image\nTotal Flux Err = {:.0e}'.format(
    ((flux - np.sum(bb_image)) / flux)))
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



(png, svg, pdf)



(png, svg, pdf)

Model Separability

Simple models have a boolean `Model.separable` property. It indicates whether the outputs are independent and is essential for computing the separability of compound models using the `is_separable()` function. Having a separable compound model means that it can be decomposed into independent models, which in turn is useful in many applications. For example, it may be easier to define inverses using the independent parts of a model than the entire model. In other cases, tools using Generalized World Coordinate System (GWCS), can be more flexible and take advantage of separable spectral and spatial transforms.

Model Sets

In some cases it is useful to describe many models of the same type but with different sets of parameter values. This could be done simply by instantiating as many instances of a `Model` as are needed. But that can be inefficient for a large number of models. To that end, all model classes in `astropy.modeling` can also be used to represent a model **set** which is a collection of models of the same type, but with different values for their parameters.

To instantiate a model set, use argument `n_models=N` where `N` is the number of models in the set when constructing the model. The value of each parameter must be a list or array of length `N`, such that each item in the array corresponds to one model in the set:

```
>>> from astropy.modeling import models
>>> g = models.Gaussian1D(amplitude=[1, 2], mean=[0, 0],
...                       stddev=[0.1, 0.2], n_models=2)
>>> print(g)
Model: Gaussian1D
Inputs: ('x',)
Outputs: ('y',)
Model set size: 2
Parameters:
    amplitude mean stddev
    --------- ---- ------
          1.0  0.0    0.1
          2.0  0.0    0.2
```

This is equivalent to two Gaussians with the parameters `amplitude=1, mean=0, stddev=0.1` and `amplitude=2, mean=0, stddev=0.2` respectively. When printing the model the parameter values are displayed as a table, with each row corresponding to a single model in the set.

The number of models in a model set can be determined using the `len` builtin:

```
>>> len(g)
2
```

Single models have a length of 1, and are not considered a model set as such.

When evaluating a model set, by default the input must be the same length as the number of models, with one input per model:

```
>>> g([0, 0.1])
array([1.        , 1.76499381])
```

The result is an array with one result per model in the set. It is also possible to broadcast a single input value to all models in the set:

```
>>> g(0)
array([1., 2.])
```

Or when the input is an array:

```
>>> x = np.array([[0, 0, 0], [0.1, 0.1, 0.1]])
>>> print(x)
[[0.  0.  0. ]
 [0.1 0.1 0.1]]
>>> g(x)
array([[1.        , 1.        , 1.        ],
       [1.76499381, 1.76499381, 1.76499381]])
```

Internally the shape of the inputs, outputs, and parameter values is controlled by an attribute - `model_set_axis`. In the above case `model_set_axis=0`:

```
>>> g.model_set_axis
0
```

This indicates that elements along the 0-th axis will be passed as inputs to individual models. Sometimes it may be useful to pass inputs along a different axis, for example the 1st axis:

```
>>> x = np.array([[0, 0, 0], [0.1, 0.1, 0.1]]).T
>>> print(x)
[[0.  0.1]
 [0.  0.1]
 [0.  0.1]]
```

Because there are two models in this model set and we are passing three inputs along the 0th axis, evaluation will fail:

```
>>> g(x)
Traceback (most recent call last):
...
ValueError: Input argument 'x' does not have the correct dimensions
in
model_set_axis=0 for a model set with n_models=2.
```

There are two ways to get around this. `model_set_axis` can be passed in when the model is evaluated:

```
>>> g(x, model_set_axis=1)
array([[1.        , 1.76499381],
       [1.        , 1.76499381],
       [1.        , 1.76499381]])
```

Or when the model is initialized:

```
>>> g = models.Gaussian1D(amplitude=[[1, 2]], mean=[[0, 0]],
...                           stddev=[[0.1, 0.2]], n_models=2,
...                           model_set_axis=1)
>>> g(x)
array([[1.        , 1.76499381],
       [1.        , 1.76499381],
       [1.        , 1.76499381]])
```

Note that in the latter case, the shape of the individual parameters has changed to 2D because now the parameters are defined along the 1st axis.

The value of `model_set_axis` is either an integer number, representing the axis along which the different parameter sets and inputs are defined, or a boolean of value `False`, in which case it indicates all model sets should use the same inputs on evaluation. For example, the above model has a value of 1 for `model_set_axis`. If `model_set_axis=False` is passed the two models will be evaluated on the same input:

```
>>> g.model_set_axis
1
>>> result = g(x, model_set_axis=False)
>>> result
array([[[1.        , 0.60653066],
        [2.        , 1.76499381]],

       [[1.        , 0.60653066],
        [2.        , 1.76499381]],
       [[1.        , 0.60653066],
        [2.        , 1.76499381]]])
>>> result[: , 0]
array([[1.        , 0.60653066],
       [1.        , 0.60653066],
       [1.        , 0.60653066]])
>>> result[: , 1]
array([[2.        , 1.76499381],
       [2.        , 1.76499381],
       [2.        , 1.76499381]])
```

Currently model sets are most useful for fitting a set of **linear** models (example) allowing a large number of models of the same type to be fitted simultaneously (and independently from each other) to some large set of inputs, such as fitting a polynomial to the time response of each pixel in a data cube. This can greatly speed up the fitting process. The speed-up is due to solving the set of equations to find the exact solution. Nonlinear models, which require an iterative algorithm, cannot be currently fit using model sets. Model sets of nonlinear models can only be evaluated.

When fitting model sets it is important that data arrays are passed to the fitter in the correct shape. The shape depends on the `model_set_axis` attribute of the model to be fit. The rule is that the index of the dependent variable that corresponds to a model set should be along the `model_set_axis` dimension. For example, for a 1D model set with 3 models with `model_set_axis == 1` the shape of `y` should be (x, 3):

```
>>> import numpy as np
>>> from astropy.modeling.models import Polynomial1D
>>> from astropy.modeling.fitting import LinearLSQFitter
>>> fitter = LinearLSQFitter()
>>> x = np.arange(4)
>>> y = np.array([2*x+1, x+4, x]).T
>>> print(y)
[[1 4 0]
 [3 5 1]
 [5 6 2]
 [7 7 3]]
>>> print(y.shape)
(4, 3)
>>> m = Polynomial1D(1, n_models=3, model_set_axis=1)
>>> mfit = fitter(m, x, y)
```

For 2D models with 3 models and `model_set_axis = 0` the shape of `z` should be (3, x, y):

```
>>> import numpy as np
>>> from astropy.modeling.models import Polynomial2D
>>> from astropy.modeling.fitting import LinearLSQFitter
>>> fitter = LinearLSQFitter()
>>> x = np.arange(8).reshape(2, 4)
>>> y = x
>>> z = np.asarray([2 * x + 1, x + 4, x + 3])
>>> print(z.shape)
(3, 2, 4)
>>> m = Polynomial2D(1, n_models=3, model_set_axis=0)
>>> mfit = fitter(m, x, y, z)
```

Model Serialization (Writing a Model to a File)

Models are serializable using the ASDF format. This can be useful in many contexts, one of which is the implementation of a Generalized World Coordinate System (GWCS).

Serializing a model to disk is possible by assigning the object to `AsdfFile.tree`:

```
>>> from asdf import AsdfFile
>>> from astropy.modeling import models
>>> rotation = models.Rotation2D(angle=23.7)
>>> f = AsdfFile()
>>> f.tree['model'] = rotation
>>> f.write_to('rotation.asdf')
```

To read the file and create the model:

```
>>> import asdf
>>> with asdf.open('rotation.asdf') as f:
...     model = f.tree['model']
>>> print(model)
Model: Rotation2D
Inputs: ('x', 'y')
Outputs: ('x', 'y')
Model set size: 1
Parameters:
    angle
    -----
     23.7
```

Compound models can also be serialized. Please note that some model attributes (e.g `meta`, `tied` parameter constraints used in fitting), as well as model sets are not yet serializable. For more information on serialization of models, see Details.

Combining Models

Basics

While the Astropy modeling package makes it very easy to define new models either from existing functions, or by writing a **`Model`** subclass, an additional way to create new models is by combining them using arithmetic expressions. This works with models built into Astropy, and most user-defined models as well. For example, it is possible to create a superposition of two Gaussians like so:

```
>>> from astropy.modeling import models
>>> g1 = models.Gaussian1D(1, 0, 0.2)
>>> g2 = models.Gaussian1D(2.5, 0.5, 0.1)
>>> g1_plus_2 = g1 + g2
```

The resulting object `g1_plus_2` is itself a new model. Evaluating, say, `g1_plus_2(0.25)` is the same as evaluating `g1(0.25) + g2(0.25)`:

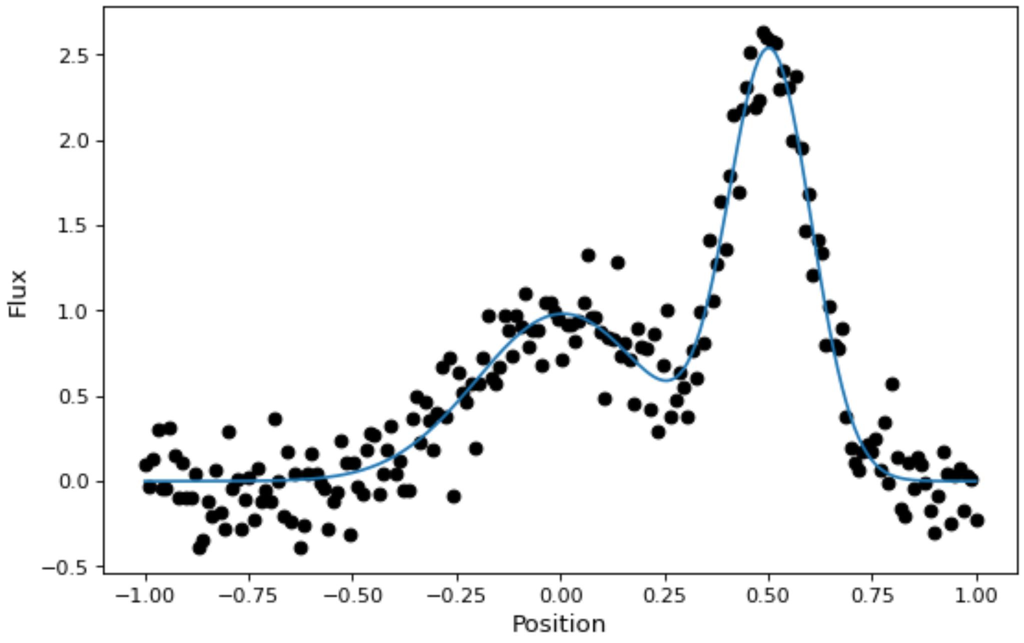
```
>>> g1_plus_2(0.25)
0.5676756958301329
>>> g1_plus_2(0.25) == g1(0.25) + g2(0.25)
True
```

This model can be further combined with other models in new expressions.

These new compound models can also be fitted to data, like most other models (though this currently requires one of the non-linear fitters):

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.modeling import models, fitting

# Generate fake data
np.random.seed(42)
g1 = models.Gaussian1D(1, 0, 0.2)
g2 = models.Gaussian1D(2.5, 0.5, 0.1)
x = np.linspace(-1, 1, 200)
y = g1(x) + g2(x) + np.random.normal(0., 0.2, x.shape)

# Now to fit the data create a new superposition with initial
# guesses for the parameters:
gg_init = models.Gaussian1D(1, 0, 0.1) + models.Gaussian1D(2, 0.5,
0.1)
fitter = fitting.SLSQPLSQFitter()
gg_fit = fitter(gg_init, x, y)

# Plot the data with the best-fit model
plt.figure(figsize=(8,5))
plt.plot(x, y, 'ko')
plt.plot(x, gg_fit(x))
plt.xlabel('Position')
plt.ylabel('Flux')
```

(png, svg, pdf)



This works for 1-D models, 2-D models, and combinations thereof, though there are some complexities involved in correctly matching up the inputs and outputs of all models used to build a compound model. You can learn more details in the Combining Models documentation.

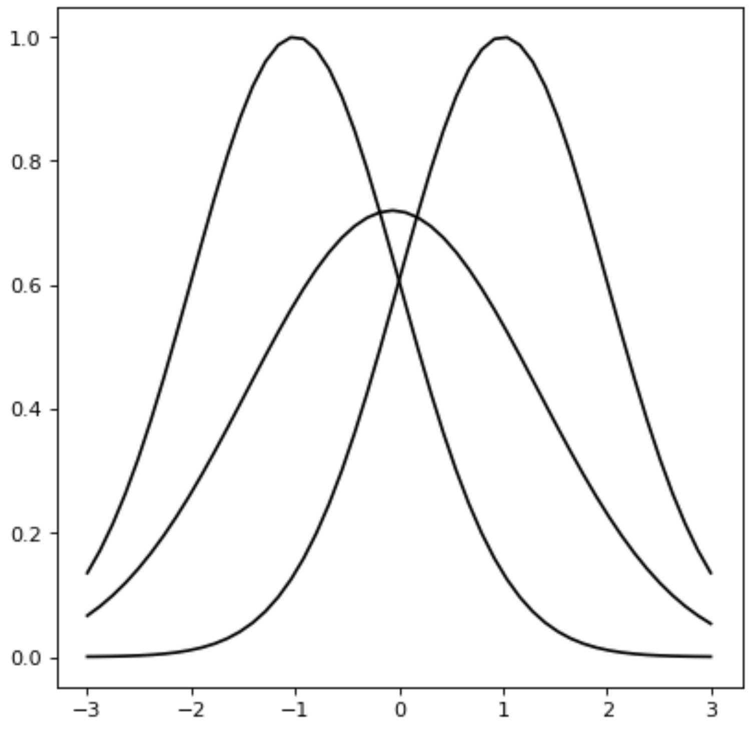
Astropy models also support convolution through the function **`convolve_models`**, which returns a compound model.

For instance, the convolution of two Gaussian functions is also a Gaussian function in which the resulting mean (variance) is the sum of the means (variances) of each Gaussian.

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.modeling import models
from astropy.convolution import convolve_models
```

```
g1 = models.Gaussian1D(1, -1, 1)
g2 = models.Gaussian1D(1, 1, 1)
g3 = convolve_models(g1, g2)

x = np.linspace(-3, 3, 50)
plt.plot(x, g1(x), 'k-')
plt.plot(x, g2(x), 'k-')
plt.plot(x, g3(x), 'k-')
```

(png, svg, pdf)



A comprehensive description

Some terminology

It is possible to create new models just by combining existing models using the arithmetic operators `+`, `-`, `*`, `/`, and `**`, or by model composition using `|` and concatenation (explained below) with `&`, as well as using **`fix_inputs()`** for reducing the number of inputs to a model.

In discussing the compound model feature, it is useful to be clear about a few terms where there have been points of confusion:

- The term “model” can refer either to a model *class* or a model *instance*.
  - All models in **`astropy.modeling`**, whether it represents some

function, a rotation, etc., are represented in the abstract by a model *class*–specifically a subclass of Model–that encapsulates the routine for evaluating the model, a list of its required parameters, and other metadata about the model.
- Per typical object-oriented parlance, a model *instance* is the object created when when calling a model class with some arguments–in most cases values for the model's parameters.

A model class, by itself, cannot be used to perform any computation because most models, at least, have one or more parameters that must be specified before the model can be evaluated on some input data. However, we can still get some information about a model class from its representation. For example:

```
>>> from astropy.modeling.models import Gaussian1D
>>> Gaussian1D
<class 'astropy.modeling.functional_models.Gaussian1D'>
Name: Gaussian1D
N_inputs: 1
N_outputs: 1
Fittable parameters: ('amplitude', 'mean', 'stddev')
```

We can then create a model *instance* by passing in values for the three parameters:

```
>>> my_gaussian = Gaussian1D(amplitude=1.0, mean=0, stddev=0.2)
>>> my_gaussian
<Gaussian1D(amplitude=1.0, mean=0.0, stddev=0.2)>
```

We now have an *instance* of Gaussian1D with all its parameters (and in principle other details like fit constraints) filled in so that we can perform calculations with it as though it were a function:

```
>>> my_gaussian(0.2)
0.6065306597126334
```

In many cases this document just refers to “models”, where the class/instance distinction is either irrelevant or clear from context. But a distinction will be made where necessary.

- A *compound model* can be created by combining two or more existing model instances which can be models that come with Astropy, user defined models, or other compound models–using Python expressions consisting of one or more of the supported binary operators. The combination of model classes is deprecated and will be removed in version 4.0.

- In some places the term *composite model* is used interchangeably with *compound model*. However, this document uses the term *composite model* to refer *only* to the case of a compound model created from the functional composition of two or more models using the pipe operator `|` as explained below. This distinction is used consistently within this document, but it may be helpful to understand the distinction.

Creating compound models

The only way to create compound models is to combine existing single models and/or compound models using expressions in Python with the binary operators `+`, `-`, `*`, `/`, `**`, `|`, and `&`, each of which is discussed in the following sections.

Warning

Creating compound models by combining classes was removed in v4.0.

The result of combining two models is a model instance:

```
>>> two_gaussians = Gaussian1D(1.1, 0.1, 0.2) + Gaussian1D(2.5, 0.5, 0.1)
>>> two_gaussians
<CompoundModel...(amplitude_0=1.1, mean_0=0.1, stddev_0=0.2, amplitude_1=2.5, mean_1=0.5, stddev_1=0.1)>
```

This expression creates a new model instance that is ready to be used for evaluation:

```
>>> two_gaussians(0.2)
0.9985190841886609
```

The `print` function provides more information about this object:

```
>>> print(two_gaussians)
Model: CompoundModel...
Inputs: ('x',)
Outputs: ('y',)
Model set size: 1
Expression: [0] + [1]
Components:
    [0]: <Gaussian1D(amplitude=1.1, mean=0.1, stddev=0.2)>

    [1]: <Gaussian1D(amplitude=2.5, mean=0.5, stddev=0.1)>
Parameters:
    amplitude_0 mean_0 stddev_0 amplitude_1 mean_1 stddev_1
    ----------- ------ -------- ----------- ------ --------
            1.1    0.1      0.2         2.5    0.5      0.1
```

There are a number of things to point out here: This model has six fittable parameters. How parameters are handled is discussed further in the section on Parameters. We also see that there is a listing of the *expression* that was used to create this compound model, which in this case is summarized as `[0] + [1]`. The `[0]` and `[1]` refer to the first and second components of the model listed next (in this case both components are the **`Gaussian1D`** objects).

Each component of a compound model is a single, non-compound model. This is the case even when including an existing compound model in a new expression. The existing compound model is not treated as a single model–instead the expression represented by that compound model is extended. An expression involving two or more compound models results in a new expression that is the concatenation of all involved models' expressions:

```
>>> four_gaussians = two_gaussians + two_gaussians
>>> print(four_gaussians)
Model: CompoundModel...
Inputs: ('x',)
Outputs: ('y',)
Model set size: 1
Expression: [0] + [1] + [2] + [3]
Components:
    [0]: <Gaussian1D(amplitude=1.1, mean=0.1, stddev=0.2)>

    [1]: <Gaussian1D(amplitude=2.5, mean=0.5, stddev=0.1)>

    [2]: <Gaussian1D(amplitude=1.1, mean=0.1, stddev=0.2)>

    [3]: <Gaussian1D(amplitude=2.5, mean=0.5, stddev=0.1)>
Parameters:
    amplitude_0 mean_0 stddev_0 amplitude_1 ... stddev_2 amplitude_3
mean_3 stddev_3
```

```
    ----------- ------ -------- ----------- ... -------- -----------
------ --------
            1.1    0.1      0.2         2.5 ...      0.2         2.5
0.5      0.1
```

Operators

Arithmetic operators

Compound models can be created from expressions that include any number of the arithmetic operators `+`, `-`, `*`, `/`, and `**`, which have the same meanings as they do for other numeric objects in Python.

> **Note**
>
> In the case of division `/` always means floating point division–integer division and the `//` operator is not supported for models).

As demonstrated in previous examples, for models that have a single output the result of evaluating a model like `A + B` is to evaluate `A` and `B` separately on the given input, and then return the sum of the outputs of `A` and `B`. This requires that `A` and `B` take the same number of inputs and both have a single output.

It is also possible to use arithmetic operators between models with multiple outputs. Again, the number of inputs must be the same between the models, as must be the number of outputs. In this case the operator is applied to the operators element-wise, similarly to how arithmetic operators work on two Numpy arrays.

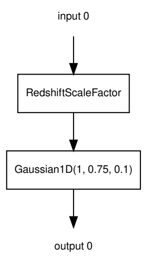
Model composition

The sixth binary operator that can be used to create compound models is the composition operator, also known as the “pipe” operator `|` (not to be confused with the boolean “or” operator that this implements for Python numeric objects). A model created with the composition operator like `M = F | G`, when evaluated, is equivalent to evaluating \(g \circ f = g(f(x))\).

> **Note**
>
> The fact that the `|` operator has the opposite sense as the functional composition operator \(\circ\) is sometimes a point of confusion. This is in part because there is no operator symbol supported in Python that corresponds well to this. The `|` operator should instead be read like the pipe operator of UNIX shell syntax: It chains together models by piping the output of the left-hand operand to the input of the right-hand operand, forming a “pipeline” of models, or transformations.

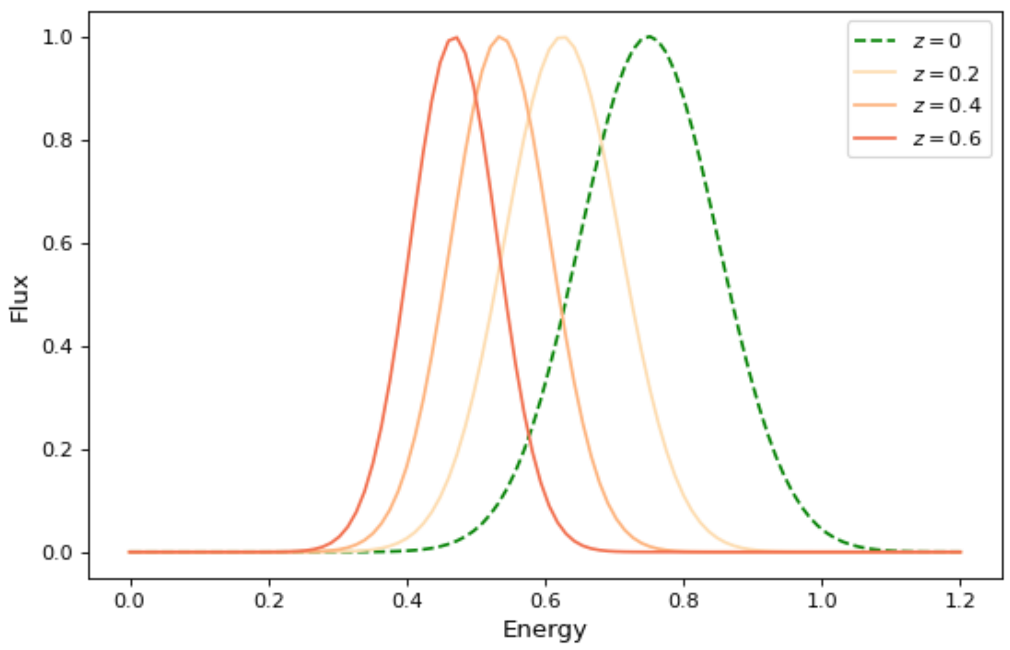
This has different requirements on the inputs/outputs of its operands than do the arithmetic operators. For composition all that is required is that the left-hand model has the same number of outputs as the right-hand model has inputs.

For simple functional models this is exactly the same as functional composition, except for the aforementioned caveat about ordering. For example, to create the following compound model:



```
import numpy as np
import matplotlib.pyplot as plt
from astropy.modeling.models import RedshiftScaleFactor, Gaussian1D

x = np.linspace(0, 1.2, 100)
g0 = RedshiftScaleFactor(0) | Gaussian1D(1, 0.75, 0.1)

plt.figure(figsize=(8, 5))
plt.plot(x, g0(x), 'g--', label='$z=0$')

for z in (0.2, 0.4, 0.6):
    g = RedshiftScaleFactor(z) | Gaussian1D(1, 0.75, 0.1)
    plt.plot(x, g(x), color=plt.cm.OrRd(z),
             label='$z={0}$'.format(z))

plt.xlabel('Energy')
plt.ylabel('Flux')
plt.legend()
```

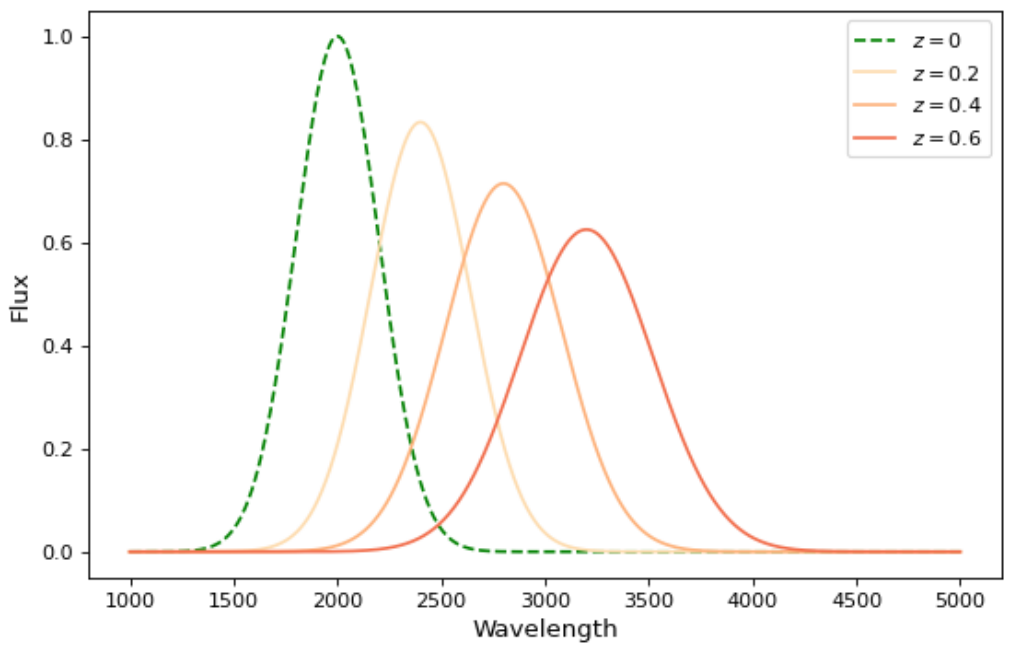
(png, svg, pdf)



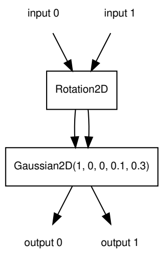
If you wish to perform redshifting in the wavelength space instead of energy, and would also like to conserve flux, here is another way to do it using model *instances*:

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.modeling.models import RedshiftScaleFactor, Gaussian1D,
Scale

x = np.linspace(1000, 5000, 1000)
g0 = Gaussian1D(1, 2000, 200)  # No redshift is same as redshift with
z=0

plt.figure(figsize=(8, 5))
plt.plot(x, g0(x), 'g--', label='$z=0$')

for z in (0.2, 0.4, 0.6):
    rs = RedshiftScaleFactor(z).inverse  # Redshift in wavelength
space
    sc = Scale(1. / (1 + z))  # Rescale the flux to conserve energy
    g = rs | g0 | sc
    plt.plot(x, g(x), color=plt.cm.OrRd(z),
             label='$z={0}$'.format(z))

plt.xlabel('Wavelength')
plt.ylabel('Flux')
plt.legend()
```

(png, svg, pdf)

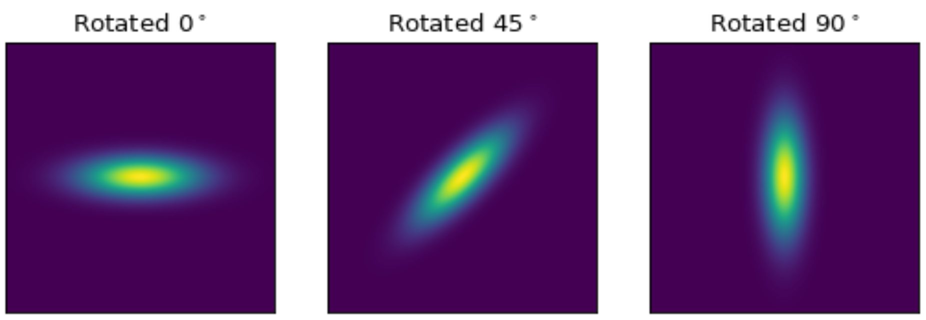


When working with models with multiple inputs and outputs the same idea applies. If each input is thought of as a coordinate axis, then this defines a pipeline of transformations for the coordinates on each axis (though it does not necessarily guarantee that these transformations are separable). For example:



```
import numpy as np
import matplotlib.pyplot as plt
from astropy.modeling.models import Rotation2D, Gaussian2D

x, y = np.mgrid[-1:1:0.01, -1:1:0.01]

plt.figure(figsize=(8, 2.5))

for idx, theta in enumerate((0, 45, 90)):
    g = Rotation2D(theta) | Gaussian2D(1, 0, 0, 0.1, 0.3)
    plt.subplot(1, 3, idx + 1)
    plt.imshow(g(x, y), origin='lower')
    plt.xticks([])
    plt.yticks([])
    plt.title('Rotated $ {0}^\circ $'.format(theta))
```

(png, svg, pdf)



Note

The above example is a bit contrived in that **Gaussian2D** already supports an optional rotation parameter. However, this demonstrates how coordinate rotation could be added to arbitrary models.

Normally it is not possible to compose, say, a model with two outputs and a function of only one input:

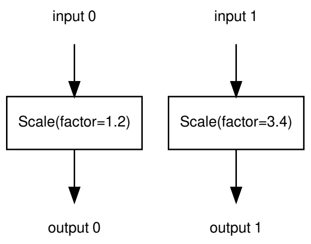
```
>>> from astropy.modeling.models import Rotation2D
>>> Rotation2D() | Gaussian1D()
Traceback (most recent call last):
...
ModelDefinitionError: Unsupported operands for |: Rotation2D
(n_inputs=2, n_outputs=2) and Gaussian1D (n_inputs=1, n_outputs=1);
n_outputs for the left-hand model must match n_inputs for the right-
hand model.
```

However, as we will see in the next section, Model concatenation, provides a means of creating models that apply transformations to only some of the outputs from a model, especially when used in concert with mappings.

Model concatenation

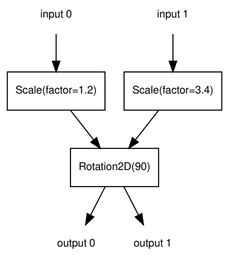
The concatenation operator `&`, sometimes also referred to as a “join”, combines two models into a single, fully separable transformation. That is, it makes a new model that takes the inputs to the left-hand model, concatenated with the inputs to the right-hand model, and returns a tuple consisting of the two models’ outputs concatenated together, without mixing in any way. In other words, it simply evaluates the two models in parallel–it can be thought of as something like a tuple of models.

For example, given two coordinate axes, we can scale each coordinate by a different factor by concatenating two **Scale** models.



```
>>> from astropy.modeling.models import Scale
>>> separate_scales = Scale(factor=1.2) & Scale(factor=3.4)
>>> separate_scales(1, 2)
(1.2, 6.8)
```

We can also combine concatenation with composition to build chains of transformations that use both “1D” and “2D” models on two (or more) coordinate axes:



```
>>> scale_and_rotate = ((Scale(factor=1.2) & Scale(factor=3.4)) |
...                     Rotation2D(90))
>>> scale_and_rotate.n_inputs
2
>>> scale_and_rotate.n_outputs
2
>>> scale_and_rotate(1, 2)
(-6.8, 1.2)
```

This is of course equivalent to an **AffineTransformation2D** with the appropriate transformation matrix:

```
>>> from numpy import allclose
>>> from astropy.modeling.models import AffineTransformation2D
>>> affine = AffineTransformation2D(matrix=[[0, -3.4], [1.2, 0]])
>>> # May be small numerical differences due to different
implementations
>>> allclose(scale_and_rotate(1, 2), affine(1, 2))
True
```

Other Topics

Model names

In the above two examples another notable feature of the generated compound model classes is that the class name, as displayed when printing the class at the command prompt, is not “TwoGaussians”, “FourGaussians”, etc. Instead it is a generated name consisting of “CompoundModel” followed by an essentially arbitrary integer that is chosen simply so that every compound model has a unique default name. This is a limitation at present, due to the limitation that it is not generally possible in Python when an object is created by an expression for it to “know” the name of the variable it will be assigned to, if any. It is possible to directly assign a name to the compound model instance by using the `Model.name` attribute:

```
>>> two_gaussians.name = "TwoGaussians"
>>> print(two_gaussians)
Model: CompoundModel...
Name: TwoGaussians
Inputs: ('x',)
Outputs: ('y',)
Model set size: 1
Expression: [0] + [1]
Components:
    [0]: <Gaussian1D(amplitude=1.1, mean=0.1, stddev=0.2)>

    [1]: <Gaussian1D(amplitude=2.5, mean=0.5, stddev=0.1)>
Parameters:
    amplitude_0 mean_0 stddev_0 amplitude_1 mean_1 stddev_1
    ----------- ------ -------- ----------- ------ --------
            1.1    0.1      0.2         2.5    0.5      0.1
```

Indexing and slicing

As seen in some of the previous examples in this document, when creating a compound model each component of the model is assigned an integer index starting from zero. These indices are assigned simply by reading the expression that defined the model, from left to right, regardless of the order of operations. For example:

```
>>> from astropy.modeling.models import Const1D
>>> A = Const1D(1.1, name='A')
>>> B = Const1D(2.1, name='B')
>>> C = Const1D(3.1, name='C')
>>> M = A + B * C
>>> print(M)
Model: CompoundModel...
Inputs: ('x',)
```

```
Outputs: ('y',)
Model set size: 1
Expression: [0] + [1] * [2]
Components:
    [0]: <Const1D(amplitude=1.1, name='A')>

    [1]: <Const1D(amplitude=2.1, name='B')>

    [2]: <Const1D(amplitude=3.1, name='C')>
Parameters:
    amplitude_0 amplitude_1 amplitude_2
    ----------- ----------- -----------
            1.1         2.1         3.1
```

In this example the expression is evaluated `(B * C) + A`–that is, the multiplication is evaluated before the addition per usual arithmetic rules. However, the components of this model are simply read off left to right from the expression `A + B * C`, with `A -> 0`, `B -> 1`, `C -> 2`. If we had instead defined `M = C * B + A` then the indices would be reversed (though the expression is mathematically equivalent). This convention is chosen for simplicity–given the list of components it is not necessary to jump around when mentally mapping them to the expression.

We can pull out each individual component of the compound model `M` by using indexing notation on it. Following from the above example, `M[1]` should return the model `B`:

```
>>> M[1]
<Const1D(amplitude=2.1, name='B')>
```

We can also take a *slice* of the compound model. This returns a new compound model that evaluates the *subexpression* involving the models selected by the slice. This follows the same semantics as slicing a **`list`** or array in Python. The start point is inclusive and the end point is exclusive. So a slice like `M[1:3]` (or just `M[1:]`) selects models `B` and `C` (and all *operators* between them). So the resulting model evaluates just the subexpression `B * C`:

```
>>> print(M[1:])
Model: CompoundModel
Inputs: ('x',)
Outputs: ('y',)
Model set size: 1
Expression: [0] * [1]
Components:
    [0]: <Const1D(amplitude=2.1, name='B')>

    [1]: <Const1D(amplitude=3.1, name='C')>
Parameters:
    amplitude_0 amplitude_1
    ----------- -----------
            2.1         3.1
```

Note

There is a change in the parameter names of a slice from versions prior to 4.0. Previously, the parameter names were identical to that of the model being sliced. Now, they are what is expected for a compound model of this type apart from the model sliced. That is, the sliced model always starts with its own relative index for its components, thus the parameter names start with a 0 suffix.

Note

Starting with 4.0, the behavior of slicing is more restrictive than previously. For example if:

```
m = m1 * m2 + m3
```

and one sliced by using `m[1:3]` previously that would return the model: `m2 + m3` even though there was never any such submodel of m. Starting with 4.0 a slice must correspond to a submodel (something that corresponds to an intermediate result of the computational chain of evaluating the compound model). So:

```
m1 * m2
```

is a submodel (i.e.,``m[:2]``) but `m[1:3]` is not. Currently this also means that in simpler expressions such as:

```
m = m1 + m2 + m3 + m4
```

where any slice should be valid in principle, only slices that include m1 are since it is part of all submodules (since the order of evaluation is:

```
((m1 + m2) + m3) + m4
```

Anyone creating compound models that wishes submodels to be available is advised to use parentheses explicitly or define intermediate models to be used in subsequent expressions so that they can be extracted with a slice or simple index depending on the context. For example, to make `m2 + m3` accessible by slice define `m` as:

```
m = m1 + (m2 + m3) + m4. In this case ``m[1:3]`` will work.
```

The new compound model for the subexpression can be evaluated like any other:

```
>>> M[1:](0)
6.51
```

Although the model `M` was composed entirely of `Const1D` models in this example, it was useful to give each component a unique name (`A`, `B`, `C`) in order to differentiate between them. This can also be used for indexing and slicing:

```
>>> print(M['B'])
Model: Const1D
Name: B
Inputs: ('x',)
Outputs: ('y',)
Model set size: 1
Parameters:
    amplitude
    ---------
          2.1
```

In this case `M['B']` is equivalent to `M[1]`. But by using the name we do not have to worry about what index that component is in (this becomes especially useful when combining multiple compound models). A current limitation, however, is that each component of a compound model must have a unique name–if some components have duplicate names then they can only be accessed by their integer index.

Slicing also works with names. When using names the start and end points are *both inclusive*:

```
>>> print(M['B':'C'])
Model: CompoundModel...
Inputs: ('x',)
Outputs: ('y',)
Model set size: 1
Expression: [0] * [1]
Components:
    [0]: <Const1D(amplitude=2.1, name='B')>
    [1]: <Const1D(amplitude=3.1, name='C')>
Parameters:
    amplitude_0 amplitude_1
    ----------- -----------
            2.1         3.1
```

So in this case `M['B':'C']` is equivalent to `M[1:3]`.

Parameters

A question that frequently comes up when first encountering compound models is how exactly all the parameters are dealt with. By now we've seen a few examples that give some hints, but a more detailed explanation is in order. This is also one of the biggest areas for possible improvements–the current behavior is meant to be practical, but is not ideal. (Some possible improvements include being able to rename parameters, and providing a means of narrowing down the number of parameters in a compound model.)

As explained in the general documentation for model parameters, every model has an attribute called **`param_names`** that contains a tuple of all the model's adjustable parameters. These names are given in a canonical order that also corresponds to the order in which the parameters should be specified when instantiating the model.

The simple scheme used currently for naming parameters in a compound model is this: The `param_names` from each component model are concatenated with each other in order from left to right as explained in the section on Indexing and slicing. However, each parameter name is appended with `_<#>`, where `<#>` is the index of the component model that parameter belongs to. For example:

```
>>> Gaussian1D.param_names
('amplitude', 'mean', 'stddev')
>>> (Gaussian1D() + Gaussian1D()).param_names
('amplitude_0', 'mean_0', 'stddev_0', 'amplitude_1', 'mean_1',
'stddev_1')
```

For consistency's sake, this scheme is followed even if not all of the components have overlapping parameter names:

```
>>> from astropy.modeling.models import RedshiftScaleFactor
>>> (RedshiftScaleFactor() | (Gaussian1D() +
Gaussian1D())).param_names
('z_0', 'amplitude_1', 'mean_1', 'stddev_1', 'amplitude_2', 'mean_2',
'stddev_2')
```

On some level a scheme like this is necessary in order for the compound model to maintain some consistency with other models with respect to the interface to its parameters. However, if one gets lost it is also possible to take advantage of indexing to make things easier. When returning a single component from a compound model the parameters associated with that component are accessible through their original names, but are still tied back to the compound model:

```
>>> a = Gaussian1D(1, 0, 0.2, name='A')
>>> b = Gaussian1D(2.5, 0.5, 0.1, name='B')
>>> m = a + b
>>> m.amplitude_0
Parameter('amplitude', value=1.0)
```

is equivalent to:

```
>>> m['A'].amplitude
Parameter('amplitude', value=1.0)
```

You can think of these both as different “views” of the same parameter. Updating one updates the other:

```
>>> m.amplitude_0 = 42
>>> m['A'].amplitude
Parameter('amplitude', value=42.0)
>>> m['A'].amplitude = 99
>>> m.amplitude_0
Parameter('amplitude', value=99.0)
```

Note, however, that the original `Gaussian1D` instance `a` has been updated:

```
>>> a.amplitude
Parameter('amplitude', value=99.0)
```

This is different than the behavior in versions prior to 4.0. Now compound model parameters share the same Parameter instance as the original model.

Advanced mappings

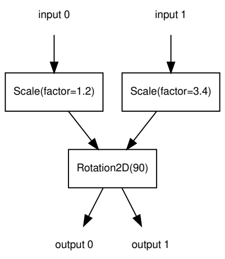
We have seen in some previous examples how models can be chained together to form a “pipeline” of transformations by using model composition and concatenation. To aid the creation of more complex chains of transformations (for example for a WCS transformation) a new class of “`mapping`” models is provided.

Mapping models do not (currently) take any parameters, nor do they perform any numeric operation. They are for use solely with the concatenation (`&`) and composition (`|`) operators, and can be used to control how the inputs and outputs of models are ordered, and how outputs from one model are mapped to inputs of another model in a composition.

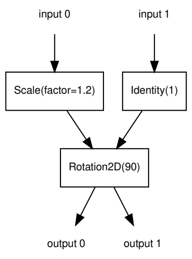
Currently there are only two mapping models: **`Identity`**, and (the somewhat generically named) **`Mapping`**.

The **`Identity`** mapping simply passes one or more inputs through, unchanged. It must be instantiated with an integer specifying the number of inputs/outputs it accepts. This can be used to trivially expand the “dimensionality” of a model in terms of the number of inputs it accepts. In the section on concatenation we saw an example like:

```
>>> m = (Scale(1.2) & Scale(3.4)) | Rotation2D(90)
```

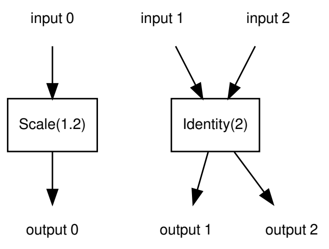


where two coordinate inputs are scaled individually and then rotated into each other. However, say we wanted to scale only one of those coordinates. It would be fine to simply use `Scale(1)` for one them, or any other model that is effectively a no-op. But that also adds unnecessary computational overhead, so we might as well simply specify that that coordinate is not to be scaled or transformed in any way. This is a good use case for **`Identity`**:



```
>>> from astropy.modeling.models import Identity
>>> m = Scale(1.2) & Identity(1)
>>> m(1, 2)
(1.2, 2.0)
```

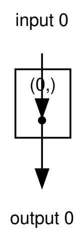
This scales the first input, and passes the second one through unchanged. We can use this to build up more complicated steps in a many-axis WCS transformation. If for example we had 3 axes and only wanted to scale the first one:



```
>>> m = Scale(1.2) & Identity(2)
>>> m(1, 2, 3)
(1.2, 2.0, 3.0)
```

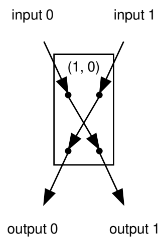
(Naturally, the last example could also be written out `Scale(1.2) & Identity(1) & Identity(1)`.)

The **Mapping** model is similar in that it does not modify any of its inputs. However, it is more general in that it allows inputs to be duplicated, reordered, or even dropped outright. It is instantiated with a single argument: a **tuple**, the number of items of which correspond to the number of outputs the **Mapping** should produce. A 1-tuple means that whatever inputs come in to the **Mapping**, only one will be output. And so on for 2-tuple or higher (though the length of the tuple cannot be greater than the number of inputs–it will not pull values out of thin air). The elements of this mapping are integers corresponding to the indices of the inputs. For example, a mapping of `Mapping((0,))` is equivalent to `Identity(1)`–it simply takes the first (0-th) input and returns it:

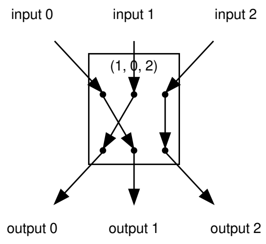


```
>>> from astropy.modeling.models import Mapping
>>> m = Mapping((0,))
>>> m(1.0)
1.0
```

Likewise `Mapping((0, 1))` is equivalent to `Identity(2)`, and so on. However, **Mapping** also allows outputs to be reordered arbitrarily:

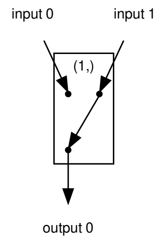


```
>>> m = Mapping((1, 0))
>>> m(1.0, 2.0)
(2.0, 1.0)
```

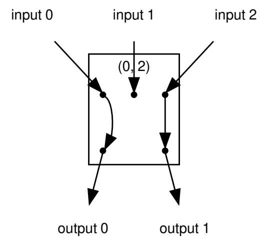


```
>>> m = Mapping((1, 0, 2))
>>> m(1.0, 2.0, 3.0)
(2.0, 1.0, 3.0)
```

Outputs may also be dropped:



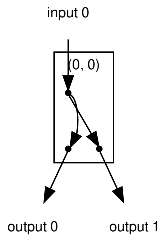
```
>>> m = Mapping((1,))
>>> m(1.0, 2.0)
2.0
```



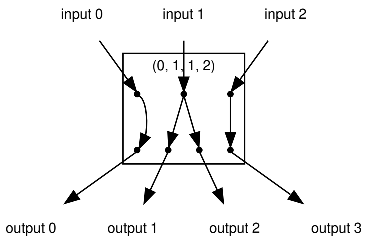
```
>>> m = Mapping((0, 2))
>>> m(1.0, 2.0, 3.0)
```

```
(1.0, 3.0)
```

Or duplicated:

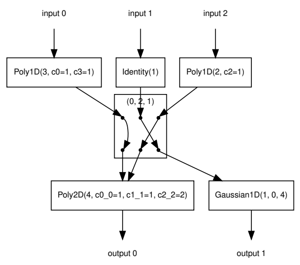


```
>>> m = Mapping((0, 0))
>>> m(1.0)
(1.0, 1.0)
```



```
>>> m = Mapping((0, 1, 1, 2))
>>> m(1.0, 2.0, 3.0)
(1.0, 2.0, 2.0, 3.0)
```

A complicated example that performs multiple transformations, some separable, some not, on three coordinate axes might look something like:



```
>>> from astropy.modeling.models import Polynomial1D as Poly1D
>>> from astropy.modeling.models import Polynomial2D as Poly2D
>>> m = ((Poly1D(3, c0=1, c3=1) & Identity(1) & Poly1D(2, c2=1)) |
...      Mapping((0, 2, 1)) |
...      (Poly2D(4, c0_0=1, c1_1=1, c2_2=2) & Gaussian1D(1, 0, 4)))
...
>>> m(2, 3, 4)
(41617.0, 0.7548396019890073)
```

This expression takes three inputs: \(x\), \(y\), and \(z\). It first takes \(x \rightarrow x^3 + 1\) and \(z \rightarrow z^2\). Then it remaps the axes so that \(x\) and \(z\) are passed in to the **Polynomial2D** to evaluate \(2x^2z^2 + xz + 1\), while simultaneously evaluating a Gaussian on \(y\). The end result is a reduction down to two coordinates. You can confirm for yourself that the result is correct.

This opens up the possibility of essentially arbitrarily complex transformation graphs. Currently the tools do not exist to make it easy to navigate and reason about highly complex compound models that use these mappings, but that is a possible enhancement for future versions.

Model Reduction

In order to save much duplication in the construction of complex models, it is possible to define one complex model that covers all cases where the variables that distinguish the models are made part of the model's input variables. The `fix_inputs` function allows defining models derived from the more complex one by setting one or more of the inputs to a constant value. Examples of this sort of situation arise when working out the transformations from detector pixel to RA, Dec, and lambda for spectrographs when the slit locations may be moved (e.g., fiber fed or commandable slit masks), or different orders may be selected (e.g., Eschelle). In the case of order, one may have a function of pixel `x`, `y`, `spectral_order` that map into `RA`, `Dec` and `lambda`. Without specifying `spectral_order`, it is ambiguous what `RA`, `Dec` and `Lambda` corresponds to a pixel location. It is usually possible to define a function of all three inputs. Presuming this model is `general_transform` then `fix_inputs` may be used to define the transform for a specific order as follows:

::

```
>>> order1_transform = fix_inputs(general_transform, {'order': 1})
```

creates a new compound model that takes only pixel position and generates `RA`, `Dec`, and `lambda`. The `fix_inputs` function can be used to set input values by position (0 is the first) or by input variable name, and more than one can be set in the dictionary supplied.

If the input model has a bounding_box, the generated model will have the bounding for the input coordinate removed.

Replace submodels

`replace_submodel()` creates a new model by replacing a submodel with a matching name with another submodel. The number of inputs and outputs of the old and new submodels should match.

```
>>> from astropy.modeling import models
>>> shift = models.Shift(-1) & models.Shift(-1)
>>> scale = models.Scale(2) & models.Scale(3)
>>> scale.name = "Scale"
>>> model = shift | scale
>>> model(2, 1)
(2.0, 0.0)
>>> new_model = model.replace_submodel('Scale', models.Rotation2D(90,
name='Rotation'))
>>> new_model(2, 1)
(6.12e-17, 1.0)
```

Parameters

Basics

Most models in this package are "parametric" in the sense that each subclass of **Model** represents an entire family of models, each member of which is distinguished by a fixed set of parameters that fit that model to some dependent and independent variable(s) (also referred to throughout the package as the outputs and inputs of the model).

Parameters are used in three different contexts within this package: Basic evaluation of models, fitting models to data, and providing information about individual models to users (including documentation).

Most subclasses of **Model**–specifically those implementing a specific physical or statistical model, have a fixed set of parameters that can be specified for instances of that model. There are a few classes of models (in particular polynomials) in which the number of parameters depends on some other property of the model (the degree in the case of polynomials).

Models maintain a list of parameter names, **param_names**. Single parameters are instances of **Parameter** which provides a proxy for the actual parameter values. Simple mathematical operations can be performed with them, but they also contain additional attributes specific to model parameters, such as any constraints on their values and documentation.

Parameter values may be scalars *or* array values. Some parameters are required by their very nature to be arrays (such as the transformation matrix for an **AffineTransformation2D**). In most other cases, however, array-valued parameters have no meaning specific to the model, and are simply combined with input arrays during model evaluation according to the standard Numpy broadcasting rules.

Parameter constraints

astropy.modeling supports several types of parameter constraints. They are implemented as properties of **Parameter**, the class which defines all fittable parameters, and can be set on individual parameters or on model instances.

The **astropy.modeling.Parameter.fixed** constraint is boolean and indicates whether a parameter is kept “fixed” or “frozen” during fitting. For example, fixing the `stddev` of a **Gaussian1D** model means it will be excluded from the list of fitted parameters:

```
>>> from astropy.modeling.models import Gaussian1D
>>> g = Gaussian1D(amplitude=10.2, mean=2.3, stddev=1.2)
>>> g.stddev.fixed
False
>>> g.stddev.fixed = True
>>> g.stddev.fixed
True
```

astropy.modeling.Parameter.bounds is a tuple of numbers setting minimum and maximum value for a parameter. `(None, None)` indicates the parameter values are not bound. `bounds` can be set also using the **min** and **max** properties. Assigning `None` to the corresponding property removes the bound on the parameter. For example, setting bounds on the `mean` value of a **Gaussian1D** model can be done either by setting `min` and `max`:

```
>>> g.mean.bounds
(None, None)
>>> g.mean.min = 2.2
>>> g.mean.bounds
(2.2, None)
>>> g.mean.max = 2.4
>>> g.mean.bounds
(2.2, 2.4)
```

or using the `bounds` property:

```
>>> g.mean.bounds = (2.2, 2.4)
```

astropy.modeling.Parameter.tied is a user supplied callable which takes a model instance and returns a value for the parameter. It is most useful with setting constraints on compounds models, for example a ratio between two parameters (example).

Constraints can also be set when the model is initialized. For example:

```
>>> g = Gaussian1D(amplitude=10.2, mean=2.3, stddev=1.2,
...                fixed={'stddev': True},
...                 bounds={'mean': (2.2, 2.4)})
>>> g.stddev.fixed
True
>>> g.mean.bounds
(2.2, 2.4)
```

Parameter examples

- Model classes can be introspected directly to find out what parameters they accept:

  ```
  >>> from astropy.modeling import models
  >>> models.Gaussian1D.param_names
  ('amplitude', 'mean', 'stddev')
  ```

  The order of the items in the `param_names` list is relevant–this is the same order in which values for those parameters should be passed in when constructing an instance of that model:

  ```
  >>> g = models.Gaussian1D(1.0, 0.0, 0.1)
  >>> g
  <Gaussian1D(amplitude=1.0, mean=0.0, stddev=0.1)>
  ```

  However, parameters may also be given as keyword arguments (in any order):

  ```
  >>> g = models.Gaussian1D(mean=0.0, amplitude=2.0, stddev=0.2)
  >>> g
  <Gaussian1D(amplitude=2.0, mean=0.0, stddev=0.2)>
  ```

  So all that really matters is knowing the names (and meanings) of the parameters that each model accepts. More information about an individual model can also be obtained using the **`help`** built-in:

  ```
  >>> help(models.Gaussian1D)
  ```

- Some types of models can have different numbers of parameters depending on other properties of the model. In particular, the parameters of polynomial models are their coefficients, the number of which depends on the polynomial's degree:

```
>>> p1 = models.Polynomial1D(degree=3, c0=1.0, c1=0.0, c2=2.0,
c3=3.0)
>>> p1.param_names
('c0', 'c1', 'c2', 'c3')
>>> p1
<Polynomial1D(3, c0=1., c1=0., c2=2., c3=3.)>
```

For the basic **Polynomial1D** class the parameters are named `c0` through `cN` where `N` is the degree of the polynomial. The above example represents the polynomial \(3x^3 + 2x^2 + 1\).

- Some models also have default values for one or more of their parameters. For polynomial models, for example, the default value of all coefficients is zero–this allows a polynomial instance to be created without specifying any of the coefficients initially:

```
>>> p2 = models.Polynomial1D(degree=4)
>>> p2
<Polynomial1D(4, c0=0., c1=0., c2=0., c3=0., c4=0.)>
```

- Parameters can then be set/updated by accessing attributes on the model of the same names as the parameters:

```
>>> p2.c4 = 1
>>> p2.c2 = 3.5
>>> p2.c0 = 2.0
>>> p2
<Polynomial1D(4, c0=2., c1=0., c2=3.5, c3=0., c4=1.)>
```

This example now represents the polynomial \(x^4 + 3.5x^2 + 2\).

- It is possible to set the coefficients of a polynomial by passing the parameters in a dictionary, since all parameters can be provided as keyword arguments:

```
>>> ch2 = models.Chebyshev2D(x_degree=2, y_degree=3)
>>> coeffs = dict((name, [idx, idx + 10])
...                  for idx, name in enumerate(ch2.param_names))
>>> ch2 = models.Chebyshev2D(x_degree=2, y_degree=3, n_models=2,
...                             **coeffs)
>>> ch2.param_sets
array([[ 0., 10.],
       [ 1., 11.],
       [ 2., 12.],
       [ 3., 13.],
       [ 4., 14.],
       [ 5., 15.],
       [ 6., 16.],
       [ 7., 17.],
       [ 8., 18.],
       [ 9., 19.],
       [10., 20.],
       [11., 21.]])
```

- Or directly, using keyword arguments:

```
>>> ch2 = models.Chebyshev2D(x_degree=2, y_degree=3,
...                          c0_0=[0, 10], c0_1=[3, 13],
...                          c0_2=[6, 16], c0_3=[9, 19],
...                          c1_0=[1, 11], c1_1=[4, 14],
...                          c1_2=[7, 17], c1_3=[10, 20,],
...                          c2_0=[2, 12], c2_1=[5, 15],
...                          c2_2=[8, 18], c2_3=[11, 21])
```

- Individual parameters values may be arrays of different sizes and shapes:

```
>>> p3 = models.Polynomial1D(degree=2, c0=1.0, c1=[2.0, 3.0],
...                          c2=[[4.0, 5.0], [6.0, 7.0], [8.0,
9.0]])
>>> p3(2.0)
array([[21., 27.],
       [29., 35.],
       [37., 43.]])
```

This is equivalent to evaluating the Numpy expression:

```
>>> import numpy as np
>>> c2 = np.array([[4.0, 5.0],
...                [6.0, 7.0],
...                [8.0, 9.0]])
>>> c1 = np.array([2.0, 3.0])
>>> c2 * 2.0**2 + c1 * 2.0 + 1.0
array([[21., 27.],
       [29., 35.],
       [37., 43.]])
```

Note that in most cases, when using array-valued parameters, the parameters must obey the standard broadcasting rules for Numpy arrays with respect to each other:

```
>>> models.Polynomial1D(degree=2, c0=1.0, c1=[2.0, 3.0],
...                     c2=[4.0, 5.0, 6.0])
Traceback (most recent call last):
...
InputParameterError: Parameter u'c1' of shape (2,) cannot be
broadcast
with parameter u'c2' of shape (3,).  All parameter arrays must have
shapes that are mutually compatible according to the broadcasting
rules.
```

Fitting Models to Data

This module provides wrappers, called Fitters, around some Numpy and Scipy fitting functions. All Fitters can be called as functions. They take an instance of **`FittableModel`** as input and modify its `parameters` attribute. The idea is to make this extensible and allow users to easily add other fitters.

Linear fitting is done using Numpy's **`numpy.linalg.lstsq`** function. There are currently two non-linear fitters which use **`scipy.optimize.leastsq`** and **`scipy.optimize.fmin_slsqp`**.

The rules for passing input to fitters are:

- Non-linear fitters currently work only with single models (not model sets).
- The linear fitter can fit a single input to multiple model sets creating multiple fitted models. This may require specifying the `model_set_axis` argument just as used when evaluating models; this may be required for the fitter to know how to broadcast the input data.
- The **`LinearLSQFitter`** currently works only with simple (not compound) models.
- The current fitters work only with models that have a single output (including bivariate functions such as **`Chebyshev2D`** but not compound models that map `x, y -> x', y'`).

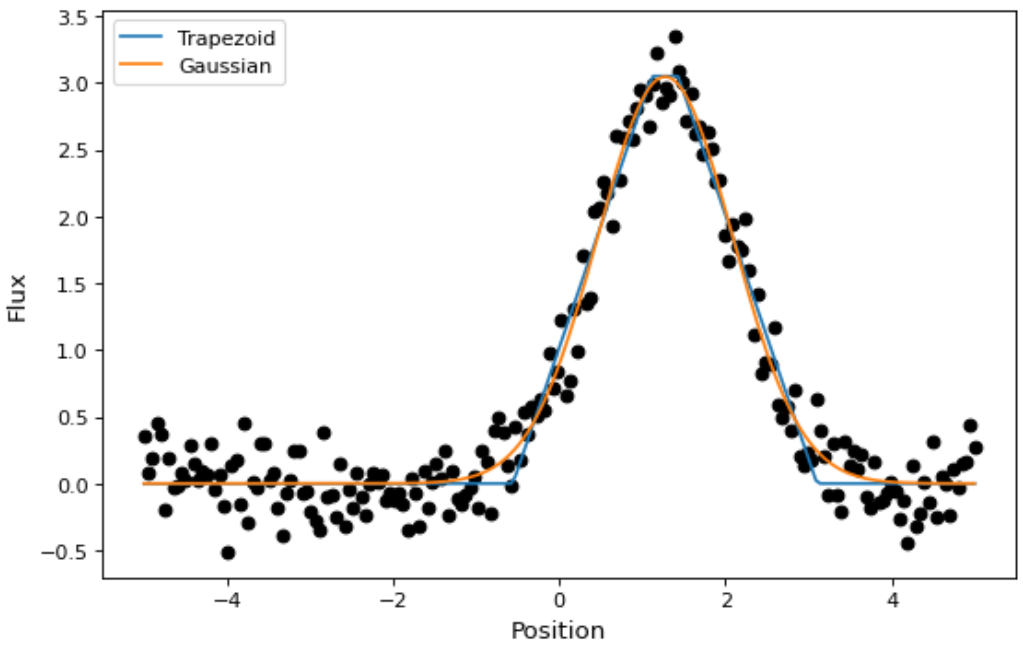
Simple 1-D model fitting

In this section, we look at a simple example of fitting a Gaussian to a simulated dataset. We use the **`Gaussian1D`** and **`Trapezoid1D`** models and the **`LevMarLSQFitter`** fitter to fit the data:

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.modeling import models, fitting
```

```
# Generate fake data
np.random.seed(0)
x = np.linspace(-5., 5., 200)
y = 3 * np.exp(-0.5 * (x - 1.3)**2 / 0.8**2)
y += np.random.normal(0., 0.2, x.shape)

# Fit the data using a box model.
# Bounds are not really needed but included here to demonstrate
usage.
t_init = models.Trapezoid1D(amplitude=1., x_0=0., width=1.,
slope=0.5,
                            bounds={"x_0": (-5., 5.)})
fit_t = fitting.LevMarLSQFitter()
t = fit_t(t_init, x, y)

# Fit the data using a Gaussian
g_init = models.Gaussian1D(amplitude=1., mean=0, stddev=1.)
fit_g = fitting.LevMarLSQFitter()
g = fit_g(g_init, x, y)

# Plot the data with the best-fit model
plt.figure(figsize=(8,5))
plt.plot(x, y, 'ko')
plt.plot(x, t(x), label='Trapezoid')
plt.plot(x, g(x), label='Gaussian')
plt.xlabel('Position')
plt.ylabel('Flux')
plt.legend(loc=2)
```

(png, svg, pdf)



As shown above, once instantiated, the fitter class can be used as a function that takes the initial model (`t_init` or `g_init`) and the data values (`x` and `y`), and returns a fitted model (`t` or `g`).

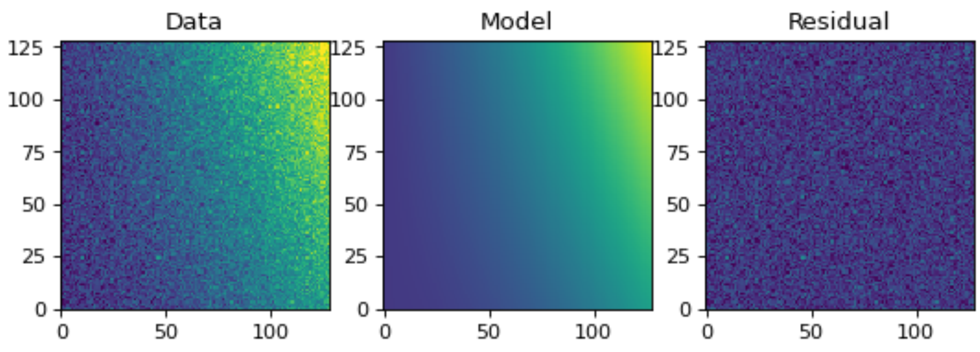
Simple 2-D model fitting

Similarly to the 1-D example, we can create a simulated 2-D data dataset, and fit a polynomial model to it. This could be used for example to fit the background in an image.

```
import warnings
import numpy as np
import matplotlib.pyplot as plt
from astropy.modeling import models, fitting

# Generate fake data
np.random.seed(0)
y, x = np.mgrid[:128, :128]
z = 2. * x ** 2 - 0.5 * x ** 2 + 1.5 * x * y - 1.
z += np.random.normal(0., 0.1, z.shape) * 50000.

# Fit the data using astropy.modeling
p_init = models.Polynomial2D(degree=2)
fit_p = fitting.LevMarLSQFitter()

with warnings.catch_warnings():
    # Ignore model linearity warning from the fitter
    warnings.simplefilter('ignore')
    p = fit_p(p_init, x, y, z)

# Plot the data with the best-fit model
plt.figure(figsize=(8, 2.5))
plt.subplot(1, 3, 1)
plt.imshow(z, origin='lower', interpolation='nearest', vmin=-1e4,
vmax=5e4)
plt.title("Data")
plt.subplot(1, 3, 2)
plt.imshow(p(x, y), origin='lower', interpolation='nearest',
vmin=-1e4,
           vmax=5e4)
plt.title("Model")
plt.subplot(1, 3, 3)
plt.imshow(z - p(x, y), origin='lower', interpolation='nearest',
vmin=-1e4,
           vmax=5e4)
```

```
plt.title("Residual")
```

(png, svg, pdf)



Support for units and quantities

Note

The functionality presented here was recently added. If you run into any issues, please don't hesitate to open an issue in the issue tracker.

The `astropy.modeling` package includes partial support for the use of units and quantities in model parameters, models, and during fitting. At this time, only some of the built-in models (such as `Gaussian1D`) support units, but this will be extended in future to all models where this is appropriate.

Setting parameters to quantities

Models can take `Quantity` objects as parameters:

```
>>> from astropy import units as u
>>> from astropy.modeling.models import Gaussian1D
>>> g1 = Gaussian1D(mean=3 * u.m, stddev=2 * u.cm, amplitude=3 *
u.Jy)
```

Accessing the parameter then returns a Parameter object that contains the value and the unit:

```
>>> g1.mean
Parameter('mean', value=3.0, unit=m)
```

It is then possible to access the individual properties of the parameter:

```
>>> g1.mean.name
'mean'
>>> g1.mean.value
3.0
>>> g1.mean.unit
Unit("m")
```

If a parameter has been initialized as a Quantity, it should always be set to a quantity, but the units don’t have to be compatible with the initial ones:

```
>>> g1.mean = 3 * u.s
>>> g1
<Gaussian1D(amplitude=3. Jy, mean=3. s, stddev=2. cm)>
```

To change the value of a parameter and not the unit, simply set the value property:

```
>>> g1.mean.value = 2
>>> g1
<Gaussian1D(amplitude=3. Jy, mean=2. s, stddev=2. cm)>
```

Setting a parameter which was originally set to a quantity to a scalar doesn’t work because it’s ambiguous whether the user means to change just the value and preserve the unit, or get rid of the unit:

```
>>> g1.mean = 2
Traceback (most recent call last):
...
UnitsError : The 'mean' parameter should be given as a Quantity
because it
was originally initialized as a Quantity
```

On the other hand, if a parameter previously defined without units is given a Quantity with a unit, this works because it is unambiguous:

```
>>> g2 = Gaussian1D(mean=3)
>>> g2.mean = 3 * u.m
```

In other words, once units are attached to a parameter, they can’t be removed due to ambiguous meaning.

Evaluating models with quantities

Quantities can be passed to model during evaluation:

```
>>> g3 = Gaussian1D(mean=3 * u.m, stddev=5 * u.cm)
>>> g3(2.9 * u.m)
<Quantity 0.1353352832366122>
>>> g3(2.9 * u.s)
Traceback (most recent call last):
...
UnitsError : Units of input 'x', s (time), could not be converted to
required input units of m (length)
```

In this case, since the mean and standard deviation have units, the value passed during evaluation also needs units:

```
>>> g3(3)
Traceback (most recent call last):
...
UnitsError : Units of input 'x', (dimensionless), could not be
converted to
required input units of m (length)
```

Equivalencies

Equivalencies require special care - a Gaussian defined in frequency space is not a Gaussian in wavelength space for example. For this reason, we don't allow equivalencies to be attached to the parameters themselves. Instead, we take the approach of converting the input data to the parameter space, and any equivalencies should be applied at evaluation time to the data (not the parameters).

Let's consider a model that is Gaussian in wavelength space:

```
>>> g4 = Gaussian1D(mean=3 * u.micron, stddev=1 * u.micron,
amplitude=3 * u.Jy)
```

By default, passing a frequency will not work:

```
>>> g4(1e2 * u.THz)
Traceback (most recent call last):
...
UnitsError : Units of input 'x', THz (frequency), could not be
converted to
required input units of micron (length)
```

But you can pass a dictionary of equivalencies to the equivalencies argument (this needs to be a dictionary since some models can contain multiple inputs):

```
>>> g4(110 * u.THz, equivalencies={'x': u.spectral()})
<Quantity 2.888986819525229 Jy>
```

The key of the dictionary should be the name of the inputs according to:

```
>>> g4.inputs
('x',)
```

It is also possible to set default equivalencies for the input parameters using the input_units_equivalencies property:

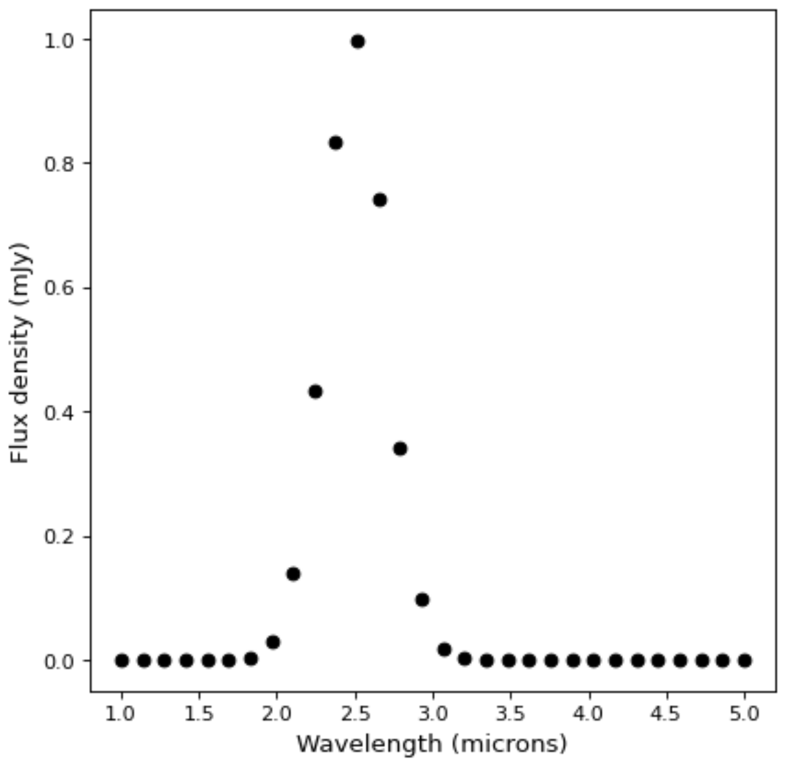
```
>>> g4.input_units_equivalencies = {'x': u.spectral()}
>>> g4(110 * u.THz)
<Quantity 2.888986819525229 Jy>
```

Fitting models with units to data

Fitting models with units to data with units should be seamless provided that the model supports fitting with units. To demonstrate this, we start off by generating synthetic data:

```
import numpy as np
from astropy import units as u
import matplotlib.pyplot as plt

x = np.linspace(1, 5, 30) * u.micron
y = np.exp(-0.5 * (x - 2.5 * u.micron)**2 / (200 * u.nm)**2) * u.mJy
plt.plot(x, y, 'ko')
plt.xlabel('Wavelength (microns)')
plt.ylabel('Flux density (mJy)')
```

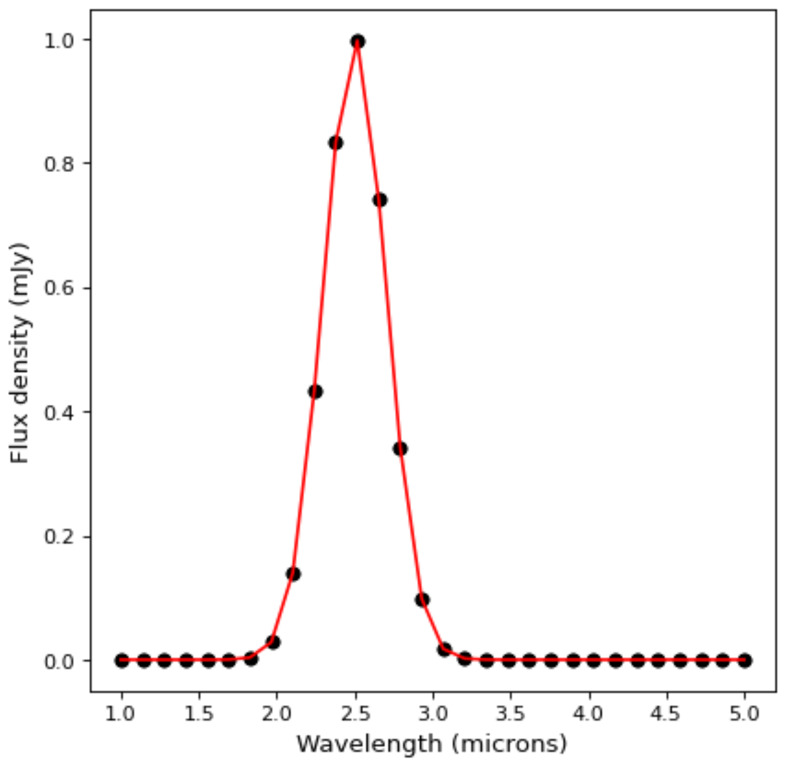
(png, svg, pdf)



and we then define the initial guess for the fitting and we carry out the fit as we would without any units:

```
from astropy.modeling import models, fitting

g5 = models.Gaussian1D(mean=3 * u.micron, stddev=1 * u.micron,
amplitude=1 * u.Jy)

fitter = fitting.LevMarLSQFitter()

g5_fit = fitter(g5, x, y)

plt.plot(x, y, 'ko')
plt.plot(x, g5_fit(x), 'r-')
plt.xlabel('Wavelength (microns)')
plt.ylabel('Flux density (mJy)')
```

(png, svg, pdf)

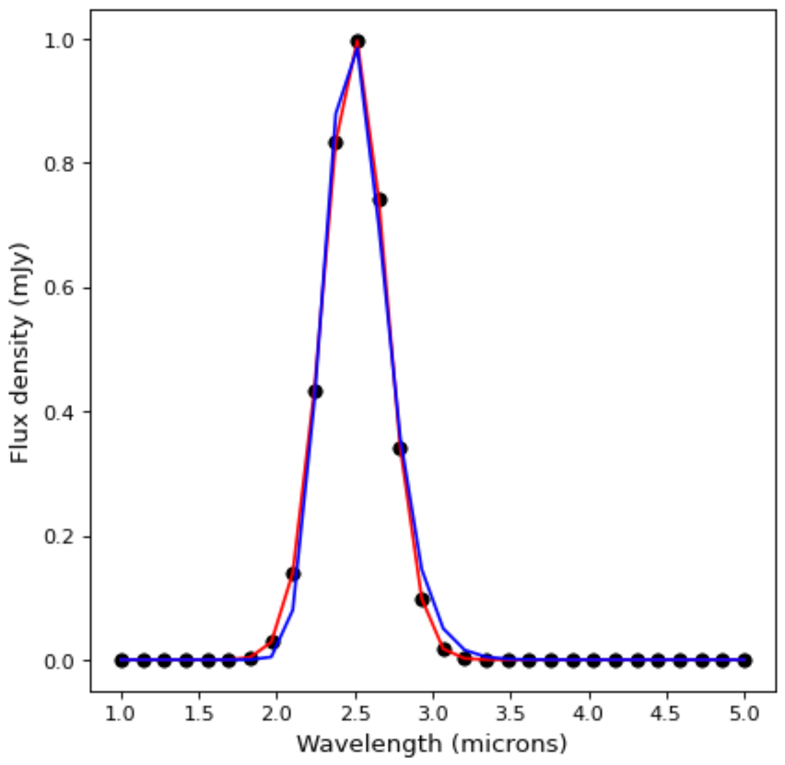


Fitting with equivalencies

Let’s now consider the case where the data is not equivalent to those of the parameters, but they are convertible via equivalencies. In this case, the equivalencies can either be passed via a dictionary as shown higher up for the evaluation examples:

```
g6 = models.Gaussian1D(mean=110 * u.THz, stddev=10 * u.THz,
amplitude=1 * u.Jy)

g6_fit = fitter(g6, x, y, equivalencies={'x': u.spectral()})

plt.plot(x, g6_fit(x, equivalencies={'x': u.spectral()}), 'b-')
plt.xlabel('Wavelength (microns)')
plt.ylabel('Flux density (mJy)')
```

(png, svg, pdf)



In this case, the fit (in blue) is slightly worse, because a Gaussian in frequency space (blue) is not a Gaussian in wavelength space (red). As mentioned previously, you can also set input_units_equivalencies on the model itself to avoid having to pass extra arguments to the fitter:

```
g6.input_units_equivalencies = {'x': u.spectral()}
g6_fit = fitter(g6, x, y)
```

Support for units in otherwise unitless models

Some models, like polynomials, do not work intrinsically with units. Instead, the **`coerce_units()`** method provides a way to add input and return units to unitless models by enclosing the unitless model with two instances of **`UnitsMapping`**. Internally the inputs are stripped of the units before passed to the model and units are attached to the result if `return_units` is specified. The method returns a new composite model:

```
>>> from astropy.modeling import models
>>> from astropy import units as u
>>> model = models.Polynomial1D(1, c0=1, c1=2)
>>> new_model = model.coerce_units(input_units={'x': u.Hz},
return_units={'y': u.s},
... input_units_equivalencies={'x':u.spectral()})
```

```
>>> new_model(10 * u.Hz)
<Quantity 21. s>
```

Changes to Modeling in v4.0

In order to make the internal code less complex, improve performance, and make the behavior of parameters in compound models more intuitive, many changes have been made internally to the modeling code, particularly to compound models and parameters. This page summarizes the important changes. More technical details are given at the end, but it is generally not necessary to read those unless you want to understand why some changes were necessary.

- Support for expressions of modeling classes has been removed. Expressions of model class instances are still fully supported. This was done to streamline the implementation, improve performance, and support the new parameter semantics. For example:

  No longer works:

  ```
  NewCompoundClass = Gaussian1D + Const1D
  ```

  Still works:

  ```
  newmodel = Gaussian1D(3.2, 7.1, 2.1) + Const1D(3.)
  ```

- Previous to v4.0, parameters were class descriptors, which meant that they could not hold values for the models. Instead, the values were held inside the models. This resulted in confusion when compound models were used since this necessitated that the compound models make copies of the values. As a result, changing the value in the compound model did not change the constituent model's parameter value and vice versa. Now parameters are distinct instances for each use and they do hold the value of the parameter, so compound models now share the same values as the constituent models.
- Previously when model sets were used, the parameter shape did not show the corresponding dimension for the number of models. Now it does. For example:

  Old:

  ```
  In [1]: g = Gaussian1D([1,1], [1,2], [2,4], n_models=2)
  In [2]: g.amplitude
  Out[2]: Parameter('amplitude', value=[1. 1.])
  In [3]: g.amplitude.shape
  Out[3]: ()
  ```

New:

```
In [1]: g = Gaussian1D([1,1], [1,2], [2,4], n_models=2)
In [2]: g.amplitude
Out[2]: Parameter('amplitude', value=[1. 1.])
In [3]: g.amplitude.shape
Out[3]: (2,)
```

- Previously the values were held in an array within the model instance and it was possible to assign values to slices of that array. Reassigning the array did update the parameters, but assigning slices does not. The new approach is to either replace the whole array by assigning to the parameters property or assign directly to the parameter value.
- The use of 'imputed' units, i.e., supplying input/output units to a compound model without them but where the component models support the `_parameter_units_for_data_units()` method is much more restricted in its applicability, which will only work when the compound expression uses the `+` or `-` operators. Past behavior led to sometimes arbitrary assignments of units, and sometimes incorrect units to the parameters.
- Slicing is more restrictive now. Previously a model defined as such:

  ```
  m = m1 * m2 + m3
  ```

  permitted this slice:

  ```
  m[1:] # results in m2 + m3
  ```

  Now, only submodels in the expression tree (think of it as the sequence of operations as performed) are permitted as slices. This means some slices that make sense do not work now. The following code illustrates what is permitted and what isn't:

  ```
  m = m1 + m2 + m3 + m4
  m[:2] # Results in m1 + m2, works
  m[1:] # Should result in m2 + m3 + m4, does not work
        # since m1 is part of all subexpressions.
  ```

- Generally, all public methods have remained unchanged. The exception is `n_submodels` that used to be a method but now is a property.
- Many of the non-public methods have changed, particularly for compound models.

- The `_CompoundModelMeta` metaclass no longer exists.

Technical Details

Parameter-related changes

Previously Python descriptors were used to define parameters. The drawback of descriptors is that all instances of the models that they are defined in share the same parameter descriptor instance. This means the parameter cannot hold different values for different model instances. The way that this was worked around was to have the model hold the actual parameter value (and any other information relevant to that particular instance of the model). The actual values of the parameters were held in an array that the model held, and that array held all values for all parameters in one array. The drawback of this approach was that compound models had to create their own array holding values for all the parameters of the compound model. As a result, the parameter values in a compound model were completely disassociated with those in the constituent model that made up the compound model. Changes to one were not reflected in the other, often leading to confusion.

The new implementation still uses attributes defined at the class level for the parameters, but only as an intermediate solution. In creating an instance of the model the class instance is used to create a local instance of a parameter that is not shared between model instances. So now parameters hold all the information directly, and can be shared between compound and constituent models.

One consequence is that the interface to fitting engines still use the model parameter array, but this array is constructed from the parameters on the fly. One may set a completely new array; the model is defined to detect such assignments and back propagate the values to the parameters. For example, when doing a fit, the fit results are propagated to the parameters in such a way. But if one assigns a slice to the model parameter array, there is no simple way to detect that assignment and make the necessary parameter updates. So this mode no longer works unless an explicit method call is made to back propagate the values. It is not recommended to use this kind of interface now.

Likewise, parameter-specific attributes also are kept by the parameter, such as the constraints on the parameter minimum or maximum, and whether it is fixed as far as fitting is concerned.

Since the parameter no longer is required to link to a specific model, it holds any dimension corresponding to the model_set size; when it previously did not.

Compound Model Implementation Changes

Compound models previously were implemented using a metaclass for the compound model while also inheriting from the Model class (which itself has a metaclass). The primary reason for this approach was to support expressions of Model classes. However, this leads to a confusing implementation, some decrease in performance, and some odd results when expressions of model classes also include model instances.

The new implementation does away with the metaclass for compound models and correspondingly no longer supports expressions of model classes, but only expressions of model instances. Previously the expression tree was a private attribute. Now the compound class is itself an expression tree.

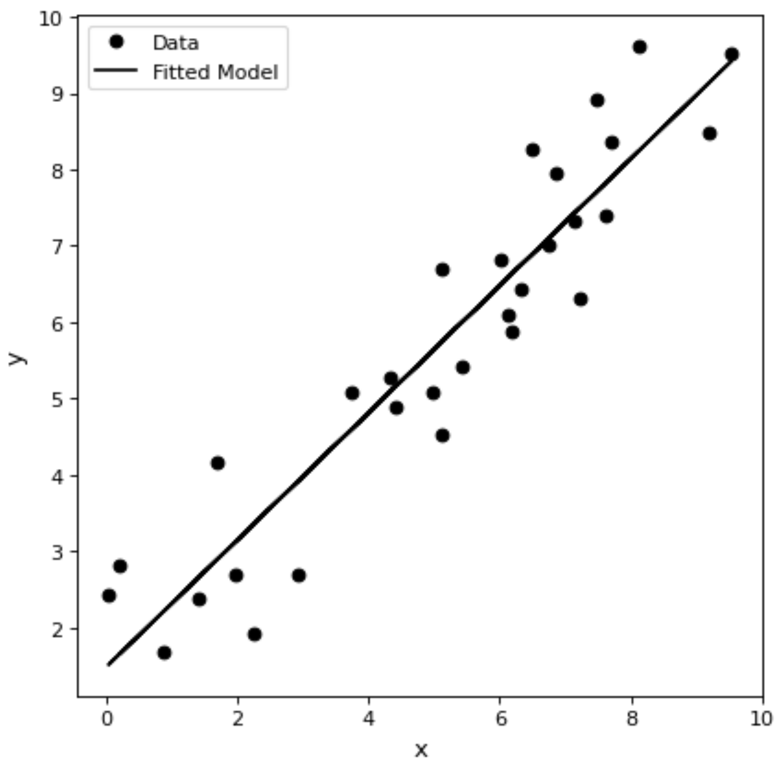
Many of the private methods of Compound Models have changed.

A Simple Example

This simple example illustrates defining a model, calculating values based on input x values, and using fitting data with a model.

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.modeling import models, fitting

# define a model for a line
line_orig = models.Linear1D(slope=1.0, intercept=0.5)

# generate x, y data non-uniformly spaced in x
# add noise to y measurements
npts = 30
np.random.seed(10)
x = np.random.uniform(0.0, 10.0, npts)
y = line_orig(x)
y += np.random.normal(0.0, 1.5, npts)

# initialize a linear fitter
fit = fitting.LinearLSQFitter()

# initialize a linear model
line_init = models.Linear1D()

# fit the data with the fitter
fitted_line = fit(line_init, x, y)

# plot the model
plt.figure()
plt.plot(x, y, 'ko', label='Data')
plt.plot(x, fitted_line(x), 'k-', label='Fitted Model')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
```

(png, svg, pdf)



Advanced Topics

Performance Tips

Initializing a compound model with many constituent models can be time consuming. If your code uses the same compound model repeatedly consider initializing it once and reusing the model.

Consider the performance tips that apply to quantities when initializing and evaluating models with quantities.

Defining New Model Classes

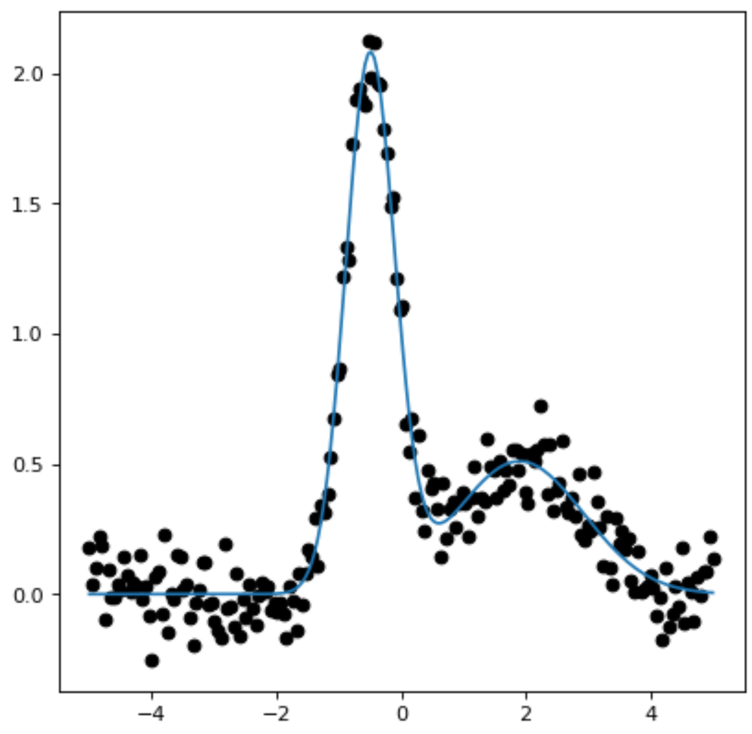
This document describes how to add a model to the package or to create a user-defined model. In short, one needs to define all model parameters and write a function which evaluates the model, that is, computes the mathematical function that implements the model. If the model is fittable, a function to compute the derivatives with respect to parameters is required if a linear fitting algorithm is to be used and optional if a non-linear fitter is to be used.

Basic custom models

For most cases, the **custom_model** decorator provides an easy way to make a new **Model** class from an existing Python callable. The following example demonstrates how to set up a model consisting of two Gaussians:

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.modeling.models import custom_model
from astropy.modeling.fitting import LevMarLSQFitter

# Define model
@custom_model
def sum_of_gaussians(x, amplitude1=1., mean1=-1., sigma1=1.,
                        amplitude2=1., mean2=1., sigma2=1.):
    return (amplitude1 * np.exp(-0.5 * ((x - mean1) / sigma1)**2) +
            amplitude2 * np.exp(-0.5 * ((x - mean2) / sigma2)**2))

# Generate fake data
np.random.seed(0)
x = np.linspace(-5., 5., 200)
m_ref = sum_of_gaussians(amplitude1=2., mean1=-0.5, sigma1=0.4,
                         amplitude2=0.5, mean2=2., sigma2=1.0)
y = m_ref(x) + np.random.normal(0., 0.1, x.shape)

# Fit model to data
m_init = sum_of_gaussians()
fit = LevMarLSQFitter()
m = fit(m_init, x, y)

# Plot the data and the best fit
plt.plot(x, y, 'o', color='k')
plt.plot(x, m(x))
```

(png, svg, pdf)



This decorator also supports setting a model's **`fit_deriv`** as well as creating models with more than one inputs. It can also be used as a normal factory function (for example `SumOfGaussians = custom_model(sum_of_gaussians)`) rather than as a decorator. See the **`custom_model`** documentation for more examples.

A step by step definition of a 1-D Gaussian model

The example described in Basic custom models can be used for most simple cases, but the following section describes how to construct model classes in general. Defining a full model class may be desirable, for example, to provide more specialized parameters, or to implement special functionality not supported by the basic **`custom_model`** factory function.

The details are explained below with a 1-D Gaussian model as an example. There are two base classes for models. If the model is fittable, it should inherit from **`FittableModel`**; if not it should subclass **`Model`**.

If the model takes parameters they should be specified as class attributes in the model's class definition using the **`Parameter`** descriptor. All arguments to the Parameter constructor are optional, and may include a default value for that parameter, a text description of the parameter (useful for **`help`** and documentation generation), as well default constraints and custom getters/setters for the parameter value. It is also possible to define a "validator" method for each parameter, enabling custom code to check whether that parameter's value is valid according to the model definition (for example if it must be non-negative). See the example in **Parameter.validator** for more details.

```
from astropy.modeling import Fittable1DModel, Parameter

class Gaussian1D(Fittable1DModel):
    n_inputs = 1
    n_outputs = 1

    amplitude = Parameter()
    mean = Parameter()
    stddev = Parameter()
```

The `n_inputs` and `n_outputs` class attributes must be integers indicating the number of independent variables that are input to evaluate the model, and the number of outputs it returns. The labels of the inputs and outputs, `inputs` and `outputs`, are generated automatically. It is possible to overwrite the default ones by assigning the desired values in the class `__init__` method, after calling `super`. `outputs` and `inputs` must be tuples of strings with length `n_outputs` and `n_inputs` respectively. Outputs may have the same labels as inputs (eg. `inputs = ('x', 'y')` and `outputs = ('x', 'y')`). However, inputs must not conflict with each other (eg. `inputs = ('x', 'x')` is incorrect) and likewise for outputs.

There are two helpful base classes in the modeling package that can be used to avoid specifying `n_inputs` and `n_outputs` for most common models. These are **Fittable1DModel** and **Fittable2DModel**. For example, the actual **Gaussian1D** model is a subclass of **Fittable1DModel**. This helps cut down on boilerplate by not having to specify `n_inputs`, `n_outputs`, `inputs` and `outputs` for many models (follow the link to Gaussian1D to see its source code, for example).

Fittable models can be linear or nonlinear in a regression sense. The default value of the **linear** attribute is `False`. Linear models should define the `linear` class attribute as `True`. Because this model is non-linear we can stick with the default.

Models which inherit from **Fittable1DModel** have the `Model._separable` property already set to `True`. All other models should define this property to indicate the Model Separability.

Next, provide methods called `evaluate` to evaluate the model and `fit_deriv`, to compute its derivatives with respect to parameters. These may be normal methods, `classmethod`, or `staticmethod`, though the convention is to use `staticmethod` when the function does not depend on any of the object's other attributes (i.e., it does not reference `self`) or any of the class's other attributes as in the case of `classmethod`. The evaluation method takes all input coordinates as separate arguments and all of the model's parameters in the same order they would be listed by `param_names`.

For this example:

```
@staticmethod
def evaluate(x, amplitude, mean, stddev):
    return amplitude * np.exp((-(1 / (2. * stddev**2)) * (x -
mean)**2))
```

It should be made clear that the `evaluate` method must be designed to take the model's parameter values as arguments. This may seem at odds with the fact that the parameter values are already available via attribute of the model (eg. `model.amplitude`). However, passing the parameter values directly to `evaluate` is a more efficient way to use it in many cases, such as fitting.

Users of your model would not generally use `evaluate` directly. Instead they create an instance of the model and call it on some input. The `__call__` method of models uses `evaluate` internally, but users do not need to be aware of it. The default `__call__` implementation also handles details such as checking that the inputs are correctly formatted and follow Numpy's broadcasting rules before attempting to evaluate the model.

Like `evaluate`, the `fit_deriv` method takes as input all coordinates and all parameter values as arguments. There is an option to compute numerical derivatives for nonlinear models in which case the `fit_deriv` method should be `None`:

```
@staticmethod
def fit_deriv(x, amplitude, mean, stddev):
    d_amplitude = np.exp(- 0.5 / stddev**2 * (x - mean)**2)
    d_mean = (amplitude *
              np.exp(- 0.5 / stddev**2 * (x - mean)**2) *
              (x - mean) / stddev**2)
    d_stddev = (2 * amplitude *
                np.exp(- 0.5 / stddev**2 * (x - mean)**2) *
                (x - mean)**2 / stddev**3)
    return [d_amplitude, d_mean, d_stddev]
```

Note that we did *not* have to define an `__init__` method or a `__call__` method for our model. For most models the `__init__` follows the same pattern, taking the parameter values as positional arguments, followed by several optional keyword arguments (constraints, etc.). The modeling framework automatically generates an `__init__` for your class that has the correct calling signature (see for yourself by calling `help(Gaussian1D.__init__)` on the example model we just defined).

There are cases where it might be desirable to define a custom `__init__`. For example, the **Gaussian2D** model takes an optional `cov_matrix` argument which can be used as an alternative way to specify the x/y_stddev and theta parameters. This is perfectly valid so long as the `__init__` determines appropriate values for the actual parameters and then calls the super `__init__` with the standard arguments. Schematically this looks something like:

```
def __init__(self, amplitude, x_mean, y_mean, x_stddev=None,
             y_stddev=None, theta=None, cov_matrix=None, **kwargs):
    # The **kwargs here should be understood as other keyword
arguments
    # accepted by the basic Model.__init__ (such as constraints)
    if cov_matrix is not None:
        # Set x/y_stddev and theta from the covariance matrix
        x_stddev = ...
        y_stddev = ...
        theta = ...

    # Don't pass on cov_matrix since it doesn't mean anything to the
base
    # class
    super().__init__(amplitude, x_mean, y_mean, x_stddev, y_stddev,
theta,
                     **kwargs)
```

Full example

```
import numpy as np
from astropy.modeling import Fittable1DModel, Parameter

class Gaussian1D(Fittable1DModel):
    amplitude = Parameter()
    mean = Parameter()
    stddev = Parameter()

    @staticmethod
    def evaluate(x, amplitude, mean, stddev):
        return amplitude * np.exp((-(1 / (2. * stddev**2)) * (x -
mean)**2))
```

```
    @staticmethod
    def fit_deriv(x, amplitude, mean, stddev):
        d_amplitude = np.exp((-(1 / (stddev**2)) * (x - mean)**2))
        d_mean = (2 * amplitude *
                  np.exp((-(1 / (stddev**2)) * (x - mean)**2)) *
                  (x - mean) / (stddev**2))
        d_stddev = (2 * amplitude *
                    np.exp((-(1 / (stddev**2)) * (x - mean)**2)) *
                    ((x - mean)**2) / (stddev**3))
        return [d_amplitude, d_mean, d_stddev]
```

A full example of a LineModel

This example demonstrates one other optional feature for model classes, which is an *inverse*. An **inverse** implementation should be a **property** that returns a new model instance (not necessarily of the same class as the model being inverted) that computes the inverse of that model, so that for some model instance with an inverse, `model.inverse(model(*input)) == input`.

```
import numpy as np
from astropy.modeling import Fittable1DModel, Parameter

class LineModel(Fittable1DModel):
    slope = Parameter()
    intercept = Parameter()
    linear = True

    @staticmethod
    def evaluate(x, slope, intercept):
        return slope * x + intercept

    @staticmethod
    def fit_deriv(x, slope, intercept):
        d_slope = x
        d_intercept = np.ones_like(x)
        return [d_slope, d_intercept]

    @property
    def inverse(self):
        new_slope = self.slope ** -1
        new_intercept = -self.intercept / self.slope
        return LineModel(slope=new_slope, intercept=new_intercept)
```

Note

The above example is essentially equivalent to the built-in **Linear1D** model.

Defining New Fitter Classes

This section describes how to add a new nonlinear fitting algorithm to this package or write a user-defined fitter. In short, one needs to define an error function and a `__call__` method and define the types of constraints which work with this fitter (if any).

The details are described below using scipy's SLSQP algorithm as an example. The base class for all fitters is **Fitter**:

```
class SLSQPFitter(Fitter):
    supported_constraints = ['bounds', 'eqcons', 'ineqcons', 'fixed',
                             'tied']

    def __init__(self):
        # Most currently defined fitters take no arguments in their
        # __init__, but the option certainly exists for custom
fitters
        super().__init__()
```

All fitters take a model (their `__call__` method modifies the model's parameters) as their first argument.

Next, the error function takes a list of parameters returned by an iteration of the fitting algorithm and input coordinates, evaluates the model with them and returns some type of a measure for the fit. In the example the sum of the squared residuals is used as a measure of fitting.:

```
def objective_function(self, fps, *args):
    model = args[0]
    meas = args[-1]
    model.fitparams(fps)
    res = self.model(*args[1:-1]) - meas
    return np.sum(res**2)
```

The `__call__` method performs the fitting. As a minimum it takes all coordinates as separate arguments. Additional arguments are passed as necessary:

```
def __call__(self, model, x, y , maxiter=MAXITER, epsilon=EPS):
    if model.linear:
            raise ModelLinearityException(
                'Model is linear in parameters; '
                'non-linear fitting methods should not be used.')
    model_copy = model.copy()
    init_values, _ = _model_to_fit_params(model_copy)
    self.fitparams = optimize.fmin_slsqp(self.errorfunc,
p0=init_values,
                                         args=(y, x),
                                         bounds=self.bounds,
                                         eqcons=self.eqcons,
                                         ineqcons=self.ineqcons)
    return model_copy
```

Defining a Plugin Fitter

`astropy.modeling` includes a plugin mechanism which allows fitters defined outside of astropy's core to be inserted into the **`astropy.modeling.fitting`** namespace through the use of entry points. Entry points are references to importable objects. A tutorial on defining entry points can be found in setuptools' documentation. Plugin fitters must to extend from the **`Fitter`** base class. For the fitter to be discovered and inserted into **`astropy.modeling.fitting`** the entry points must be inserted into the **`astropy.modeling`** entry point group

```
setup(
      # ...
      entry_points = {'astropy.modeling': 'PluginFitterName =
fitter_module:PlugFitterClass'}
)
```

This would allow users to import the `PlugFitterName` through **`astropy.modeling.fitting`** by

```
from astropy.modeling.fitting import PlugFitterName
```

One project which uses this functionality is Saba and be can be used as a reference.

Using a Custom Statistic Function

This section describes how to write a new fitter with a user-defined statistic function. The example below shows a specialized class which fits a straight line with uncertainties in both variables.

The following import statements are needed:

```
import numpy as np
from astropy.modeling.fitting import (_validate_model,
                                      _fitter_to_model_params,
                                      _model_to_fit_params, Fitter,
                                      _convert_input)
from astropy.modeling.optimizers import Simplex
```

First one needs to define a statistic. This can be a function or a callable class.:

```
def chi_line(measured_vals, updated_model, x_sigma, y_sigma, x):
    """
    Chi^2 statistic for fitting a straight line with uncertainties in
x and
    y.

    Parameters
    ----------
    measured_vals : array
    updated_model : `~astropy.modeling.ParametricModel`
        model with parameters set by the current iteration of the
optimizer
    x_sigma : array
        uncertainties in x
    y_sigma : array
        uncertainties in y

    """
    model_vals = updated_model(x)
    if x_sigma is None and y_sigma is None:
        return np.sum((model_vals - measured_vals) ** 2)
    elif x_sigma is not None and y_sigma is not None:
        weights = 1 / (y_sigma ** 2 + updated_model.parameters[1] **
2 *
                       x_sigma ** 2)
        return np.sum((weights * (model_vals - measured_vals)) ** 2)
    else:
        if x_sigma is not None:
            weights = 1 / x_sigma ** 2
        else:
            weights = 1 / y_sigma ** 2
        return np.sum((weights * (model_vals - measured_vals)) ** 2)
```

In general, to define a new fitter, all one needs to do is provide a statistic function and an optimizer. In this example we will let the optimizer be an optional argument to the fitter and will set the statistic to `chi_line` above:

```
class LineFitter(Fitter):
    """
    Fit a straight line with uncertainties in both variables

    Parameters
    ----------
    optimizer : class or callable
        one of the classes in optimizers.py (default: Simplex)
    """

    def __init__(self, optimizer=Simplex):
        self.statistic = chi_line
        super().__init__(optimizer, statistic=self.statistic)
```

The last thing to define is the `__call__` method:

```
def __call__(self, model, x, y, x_sigma=None, y_sigma=None,
**kwargs):
    """
    Fit data to this model.

    Parameters
    ----------
    model : `~astropy.modeling.core.ParametricModel`
        model to fit to x, y
    x : array
        input coordinates
    y : array
        input coordinates
    x_sigma : array
        uncertainties in x
    y_sigma : array
        uncertainties in y
    kwargs : dict
        optional keyword arguments to be passed to the optimizer

    Returns
    ------
    model_copy : `~astropy.modeling.core.ParametricModel`
        a copy of the input model with parameters set by the fitter

    """
    model_copy = _validate_model(model,

self._opt_method.supported_constraints)
```

```
    farg = _convert_input(x, y)
    farg = (model_copy, x_sigma, y_sigma) + farg
    p0, _ = _model_to_fit_params(model_copy)

    fitparams, self.fit_info = self._opt_method(
        self.objective_function, p0, farg, **kwargs)
    _fitter_to_model_params(model_copy, fitparams)

    return model_copy
```

Adding support for units in a model (Advanced)

Evaluation

To make it so that your models can accept parameters with units and be evaluated using inputs with units, you need to make sure that the **evaluate()** method works correctly with input values and parameters with units. For simple arithmetic, this may work out of the box since **Quantity** objects are understood by a number of Numpy functions.

If users of your models provide input during evaluation that is not compatible with the parameter units, they may get cryptic errors such as:

```
UnitsError : Can only apply 'subtract' function to dimensionless
quantities
when other argument is not a quantity (unless the latter is all
zero/infinity/nan)
```

There are several attributes or properties that can be set on models that adjust the behavior of models with units. These attributes can be changed from the defaults in the class definition, e.g.:

```
class MyModel(Model):
    input_units = {'x': u.deg}
    ...
```

Note that these are all optional.

`input_units`

You can easily add checking of the input units by adding an `input_units` property or attribute on your model class. This should return either **None** (to indicate no constraints) or a dictionary where the keys are the input names (e.g.

`x` for many 1D models) and the values are the units expected, which can be a function of the parameter units:

```
@property
def input_units(self):
    if self.mean.unit is None:
        return None
    else:
        return {'x': self.mean.unit}
```

If the user then gives values with incorrect input units, a clear error will be displayed:

```
UnitsError: Units of input 'x', (dimensionless), could not be
converted to
required input units of m (length)
```

Note that the input units don't have to match exactly those returned by `input_units`, but be convertible to them. In addition, `input_units` can also be specified as an attribute rather than a property in simple cases:

```
input_units = {'x': u.deg}
```

`return_units`
Similarly to input_units, this should be dictionary that maps the return values of a model to units. If **`evaluate()`** was called with quantities but returns unitless values, the units are added to the output. If the return values are quantities in different units, they are converted to `return_units`.

`input_units_strict`
If set to **`True`**, values that are passed in compatible units will be converted to the exact units specified in `input_units`.

This attribute can also be a dictionary that maps input names to a Boolean to enable converting of that input to the specified unit.

`input_units_equivalencies`
This can be set to a dictionary that maps the input names to a list of equivalencies, for example:

```
input_units_equivalencies = {'nu': u.spectral()}
```

`_input_units_allow_dimensionless`
If set to **`True`**, values that are plain scalars or Numpy arrays can be passed to

evaluate even if `input_units` specifies that the input should have units. It is up to the **`evaluate()`** to then decide how to handle these dimensionless values. This attribute can also be a dictionary that maps input names to a Boolean to enable passing dimensionless values to **`evaluate()`** for that input.

Fitting

To allow models with parameters that have units to be fitted to data with units, you will need to add a method called `_parameter_units_for_data_units` to your model class. This should take two arguments `input_units` and `output_units` - `input_units` will be set to a dictionary with the units of the independent variables in the data, while `output_units` will be set to a dictionary with the units the dependent variables in the data (for example, for a simple 1D model, `input_units` will have one key, `x`, and `output_units` will have one key, `y`). This method should then return a dictionary giving for each parameter the units the parameter should be converted to so that the model could be used on the data if units were removed from both the models and the data. The following example shows the implementation for the 1D Gaussian:

```
def _parameter_units_for_data_units(self, inputs_unit, outputs_unit):
    return {'mean': inputs_unit['x'],
            'stddev': inputs_unit['x'],
            'amplitude': outputs_unit['y']}
```

With this method in place, the model can then be fit to data that has units.

Pre-Defined Models

Some of the pre-defined models are listed and illustrated.

1D Models

Operations

These models perform simple mathematical operations.

- **`Const1D`** model returns the constant replicated by the number of input x values.
- **`Multiply`** model multiples the input x values by a factor and propagates

units if the factor is a **Quantity**.

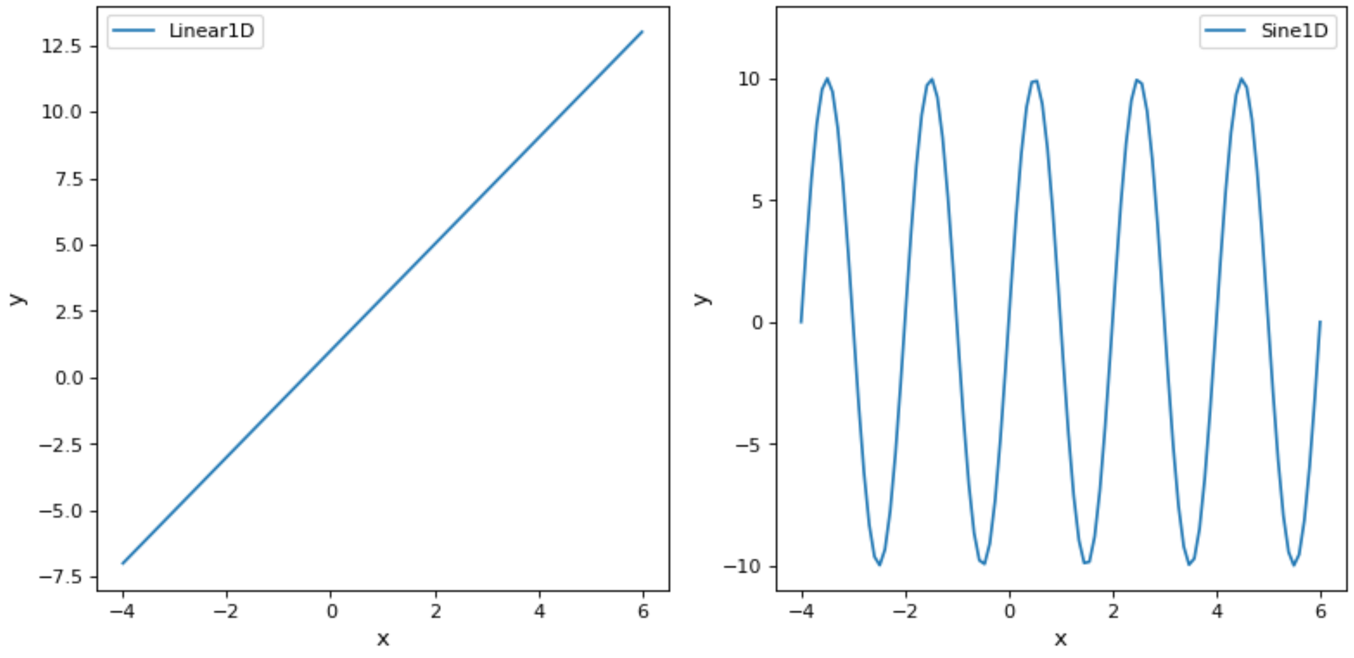
- **RedshiftScaleFactor** model multiples the input x values by a (1 + z) factor.
- **Scale** model multiples by a factor without changing the units of the result.
- **Shift** model adds a constant to the input x values.

Shapes

These models provide shapes, often used to model general x, y data.

- **Linear1D** model provides a line parameterizied by the slope and y-intercept
- **Sine1D** model provides a sine parameterized by an amplitude, frequency, and phase.

(png, svg, pdf)



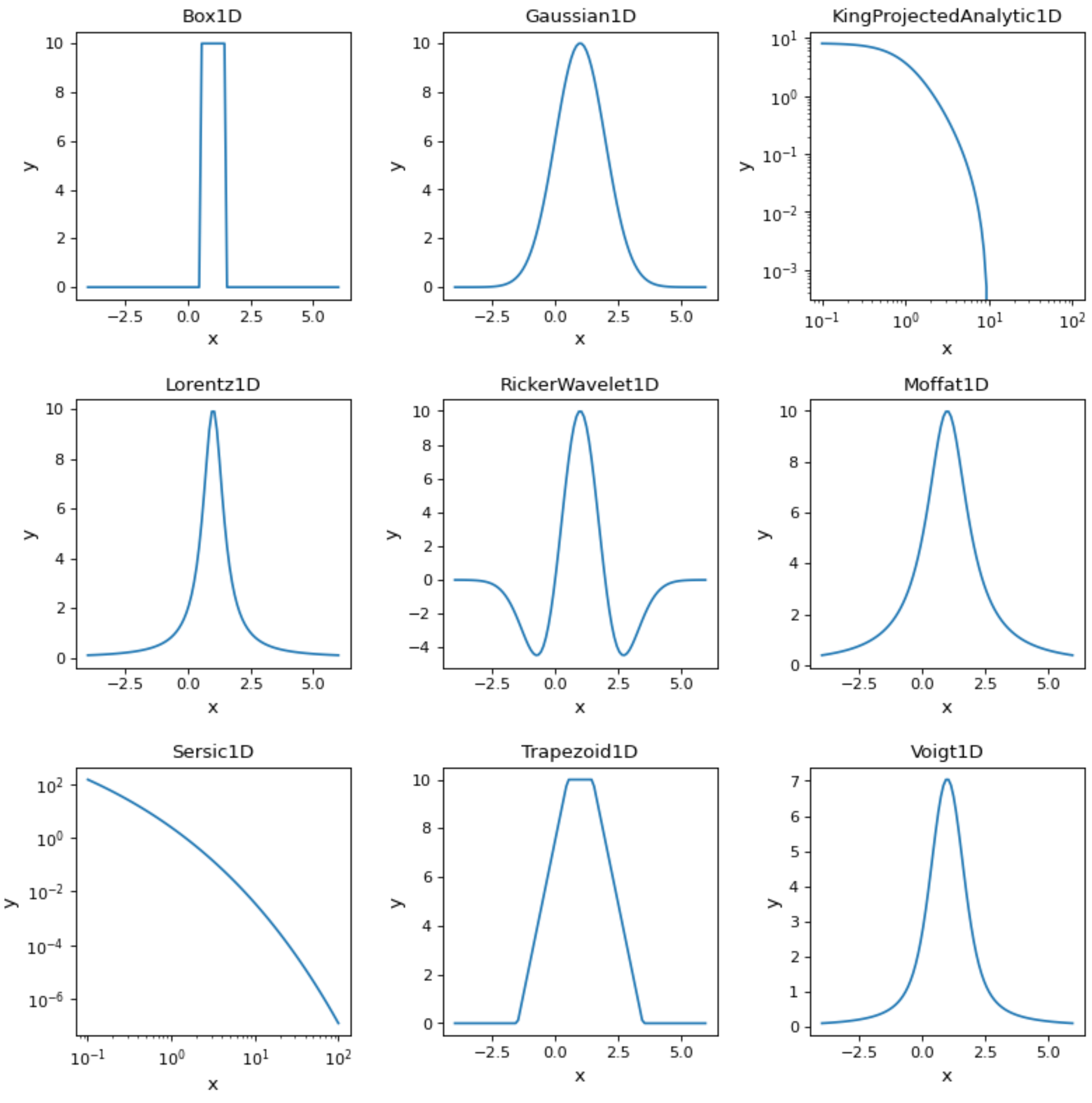
Profiles

These models provide profiles, often used for lines in spectra.

- **Box1D** model computes a box function with an amplitude centered at x_0 with the specified width.
- **Gaussian1D** model computes a Gaussian with an amplitude centered at x_0 with the specified width.

- `KingProjectedAnalytic1D` model computes the analytic form of the a King model with an amplitude and core and tidal radii.
- `Lorentz1D` model computes a Lorentzian with an amplitude centered at x_0 with the specified width.
- `RickerWavelet1D` model computes a RickerWavelet function with an amplitude centered at x_0 with the specified width.
- `Moffat1D` model computes a Moffat function with an amplitude centered at x_0 with the specified width.
- `Sersic1D` model computes a Sersic model with an amplitude with an effective radius and the specified sersic index.
- `Trapezoid1D` model computes a box with sloping sides with an amplitude centered at x_0 with the specified width and sides wit the specified slope.
- `Voigt1D` model computes a Voigt function with an amplitude centered at x_0 with the specified Lorentzian and Gaussian widths.

(png, svg, pdf)



2D Models

These models take as input x and y arrays.

Operations

These models perform simple mathematical operations.

- **`Const2D`** model returns the constant replicated by the number of input x and y values.

Shapes

These models provide shapes, often used to model general x, y, z data.

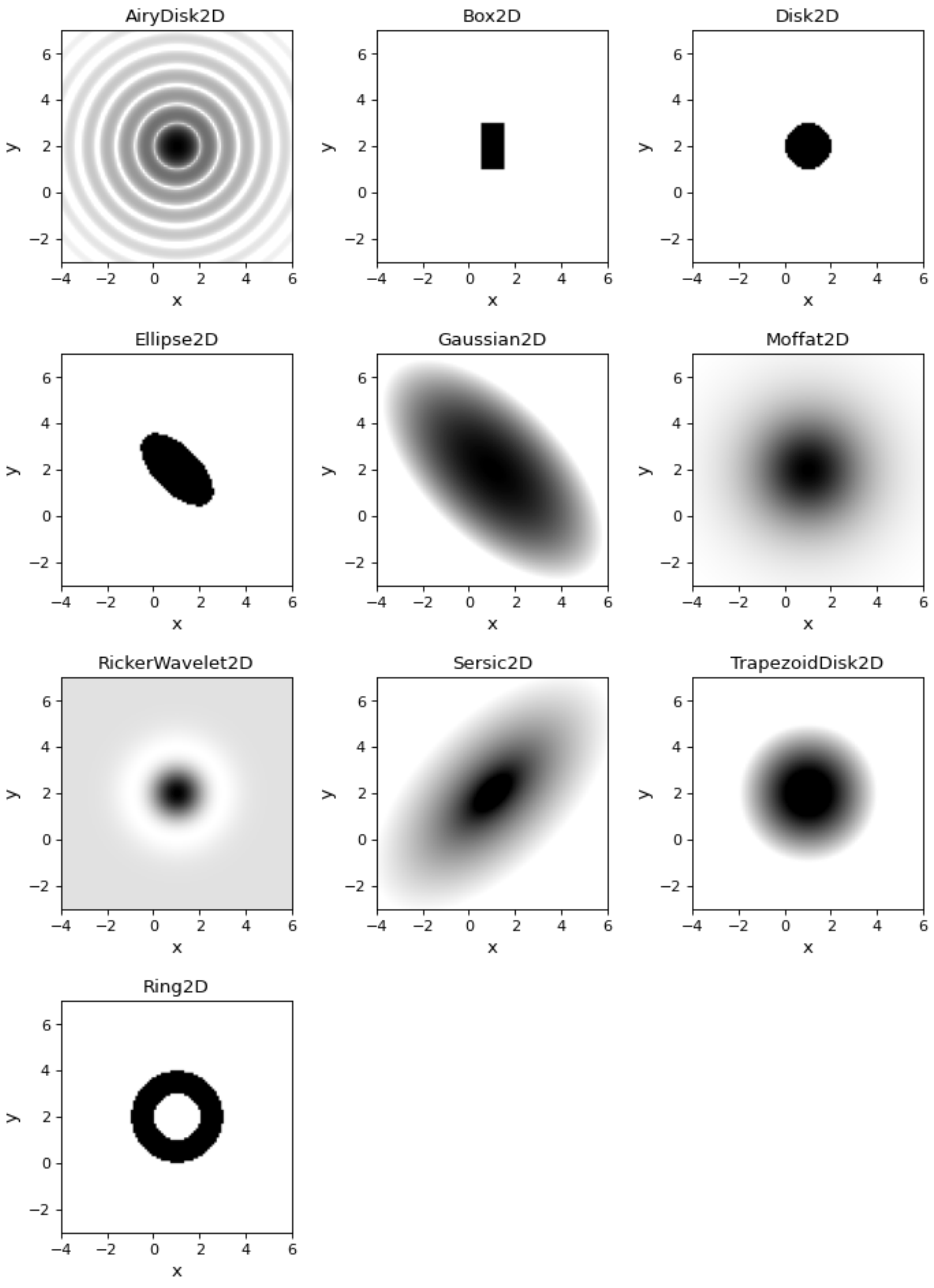
- `Planar2D` model computes a tilted plan with specified x,y slopes and z intercept

Profiles

These models provide profiles, often used sources in images. All models have parameters giving the x,y location of the center and an amplitude.

- `AiryDisk2D` model computes the Airy function for a radius
- `Box2D` model computes a box with x,y dimensions
- `Disk2D` model computes a disk a radius
- `Ellipse2D` model computes an ellipse with major and minor axis and rotation angle
- `Gaussian2D` model computes a Gaussian with x,y standard deviations and rotation angle
- `Moffat2D` model computes a Moffat with x,y dimensions and alpha (power index) and gamma (core width)
- `RickerWavelet2D` model computes a symmetric RickerWavelet function with the specified sigma
- `Sersic2D` model computes a Sersic profile with an effective half-light radius, rotation, and Sersic index
- `TrapezoidDisk2D` model computes a disk with a radius and slope
- `Ring2D` model computes a ring with inner and outer radii

(png, svg, pdf)



Physical Models

These are models that are physical motivated, generally as solutions to physical problems. This is in contrast to those that are mathematically motivated, generally as solutions to mathematical problems.

BlackBody

The **BlackBody** model provides a model for using Planck’s Law. The blackbody function is

\[B_{\nu}(T) = A \frac{2 h \nu^{3} / c^{2}}{exp(h \nu / k T) - 1}\]

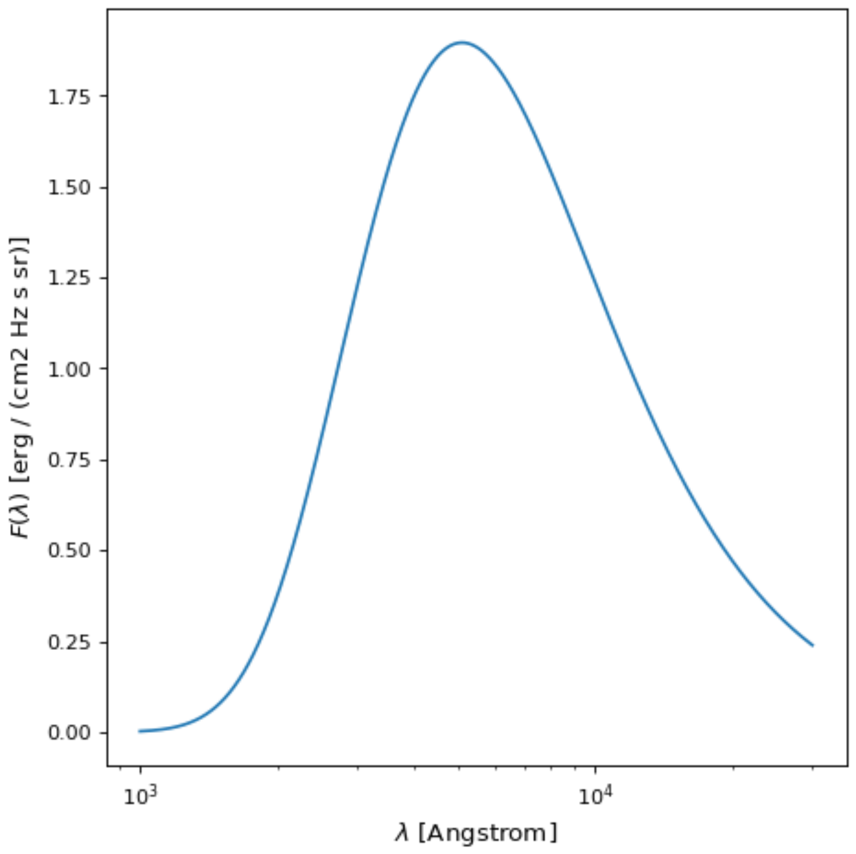
where \(\nu\) is the frequency, \(T\) is the temperature, \(A\) is the scaling factor, \(h\) is the Plank constant, \(c\) is the speed of light, and \(k\) is the Boltzmann constant.

The two parameters of the model the scaling factor `scale` (A) and the absolute temperature `temperature` (T). If the `scale` factor does not have units, then the result is in units of spectral radiance, specifically ergs/(cm^2 Hz s sr). If the `scale` factor is passed with spectral radiance units, then the result is in those units (e.g., ergs/(cm^2 A s sr) or MJy/sr). Setting the `scale` factor with units of ergs/(cm^2 A s sr) will give the Planck function as \(B_\lambda\). The temperature can be passed as a Quantity with any supported temperature unit.

An example plot for a blackbody with a temperature of 10000 K and a scale of 1 is shown below. A scale of 1 shows the Planck function with no scaling in the default units returned by **BlackBody**.

```
import numpy as np
import matplotlib.pyplot as plt

from astropy.modeling.models import BlackBody
import astropy.units as u

wavelengths = np.logspace(np.log10(1000), np.log10(3e4), num=1000) *
u.AA

# blackbody parameters
temperature = 10000 * u.K

# BlackBody provides the results in ergs/(cm^2 Hz s sr) when scale
has no units
bb = BlackBody(temperature=temperature, scale=10000.0)
bb_result = bb(wavelengths)

fig, ax = plt.subplots(ncols=1)
ax.plot(wavelengths, bb_result, '-')
ax.set_xscale('log')
ax.set_xlabel(r"$\lambda$ [{}]".format(wavelengths.unit))
ax.set_ylabel(r"$F(\lambda)$ [{}]".format(bb_result.unit))

plt.tight_layout()
plt.show()
```

(png, svg, pdf)



The **bolometric_flux()** member function gives the bolometric flux using \(\sigma T^4/\pi\) where \(\sigma\) is the Stefan-Boltzmann constant.

The **lambda_max()** and **nu_max()** member functions give the wavelength and frequency of the maximum for \(B_\lambda\) and \(B_\nu\), respectively, calculated using Wein's Law.

Note

Prior to v4.0, the `BlackBody1D` and the functions `blackbody_nu` and `blackbody_lambda` were provided. `BlackBody1D` was a more limited blackbody model that was specific to integrated fluxes from sources. The capabilities of all three can be obtained with **BlackBody**. See Blackbody Module (deprecated capabilities) and astropy issue #9066 for details.

Drude1D

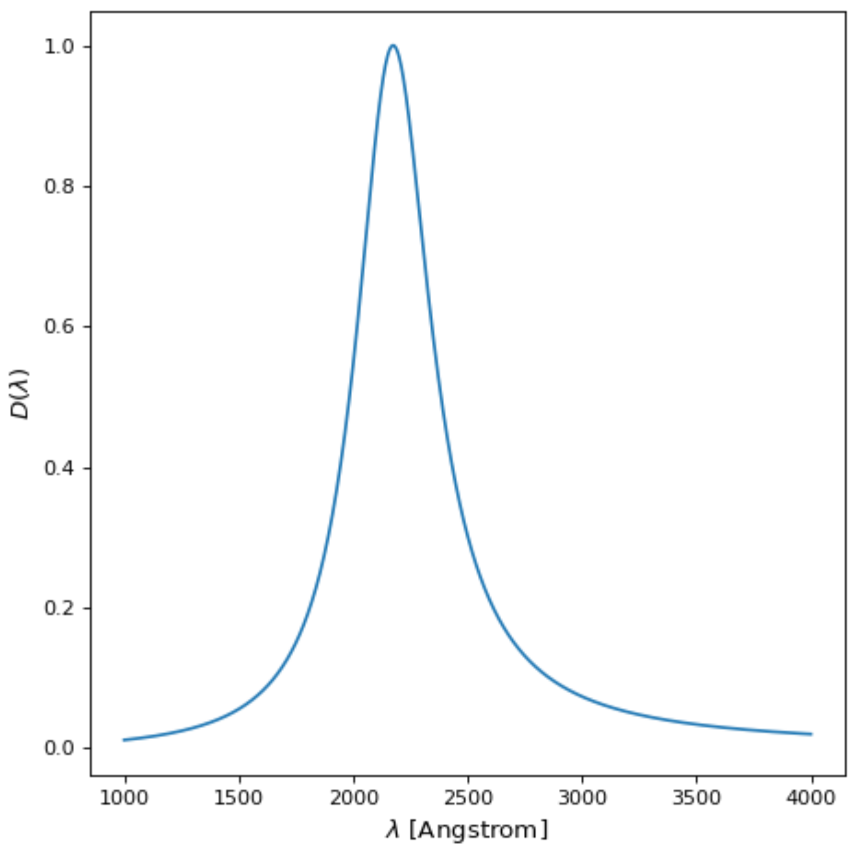
The **`Drude1D`** model provides a model for the behavior of an electron in a material (see Drude Model). Like the **`Lorentz1D`** model, the Drude model has broader wings than the **`Gaussian1D`** model. The Drude profile has been used to model dust features including the 2175 Angstrom extinction feature and the mid-infrared aromatic/PAH features. The Drude function at x is

$$D(x) = A \frac{(f/x_0)^2}{((x/x_0 - x_0/x)^2 + (f/x_0)^2}$$

where A is the amplitude, f is the full width at half maximum, and x_0 is the central wavelength. An example of a Drude1D model with $x_0 = 2175$ Angstrom and $f = 400$ Angstrom is shown below.

```
import numpy as np
import matplotlib.pyplot as plt

from astropy.modeling.models import Drude1D
import astropy.units as u

wavelengths = np.linspace(1000, 4000, num=1000) * u.AA

# Parameters and model
mod = Drude1D(amplitude=1.0, x_0=2175. * u.AA, fwhm=400. * u.AA)
mod_result = mod(wavelengths)

fig, ax = plt.subplots(ncols=1)
ax.plot(wavelengths, mod_result, '-')

ax.set_xlabel(r"$\lambda$ [{}]".format(wavelengths.unit))
ax.set_ylabel(r"$D(\lambda)$")

plt.tight_layout()
plt.show()
```

(png, svg, pdf)



NFW

The **NFW** model computes a 1-dimensional Navarro–Frenk–White profile. The dark matter density in an NFW profile is given by:

$$\rho(r)=\frac{\delta_c\rho_{c}}{r/r_s(1+r/r_s)^2}$$

where ρ_{c} is the critical density of the Universe at the redshift of the profile, δ_c is the over density, and r_s is the scale radius of the profile.

This model relies on three parameters:

- `mass` : the mass of the profile (in solar masses if no units are provided)
- `concentration` : the profile concentration
- `redshift` : the redshift of the profile

As well as two optional initialization variables:

- `massfactor` : tuple or string specifying the overdensity type and factor (default (“critical”, 200)) `cosmo` : the cosmology for density calculation (default default_cosmology)

Note

Initialization of NFW profile object required before evaluation (in order to set mass overdensity and cosmology).

Sample plots of an NFW profile with the following parameters are displayed below:

`mass` = \(2.0 x 10^{15} M_{sun}\)

`concentration` = 8.5

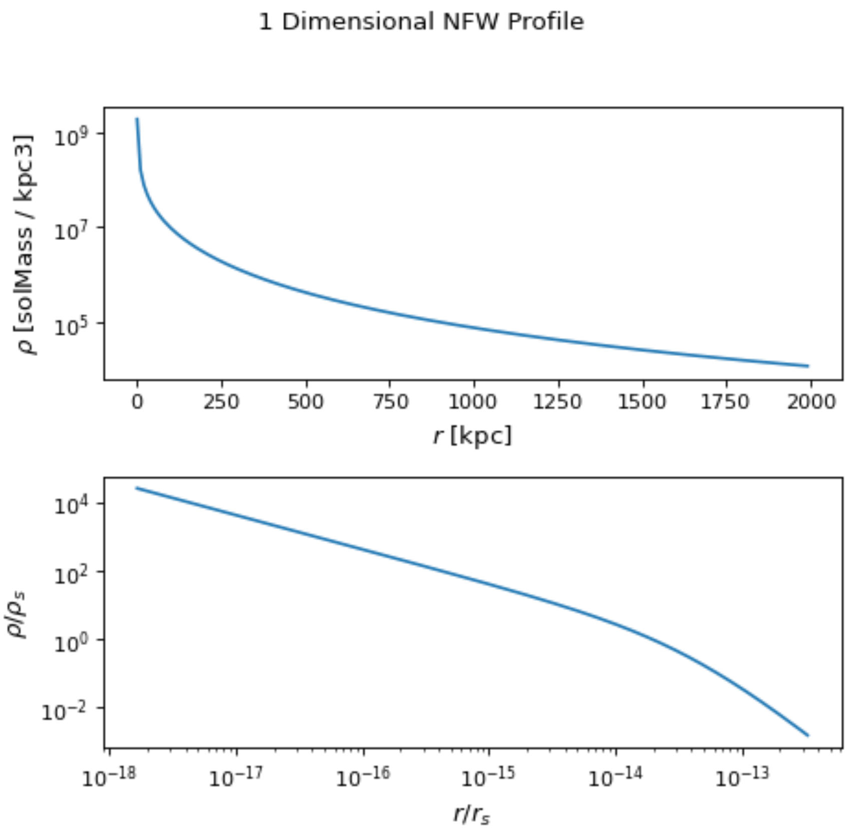
`redshift` = 0.63

The first plot is of the NFW profile density as a function of radius. The second plot displays the profile density and radius normalized by the NFW scale density and scale radius, respectively. The scale density and scale radius are available as attributes `rho_s` and `r_s`, and the overdensity radius can be accessed via `r_virial`.

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.modeling.models import NFW
import astropy.units as u
from astropy import cosmology

# NFW Parameters
mass = u.Quantity(2.0E15, u.M_sun)
concentration = 8.5
redshift = 0.63
cosmo = cosmology.Planck15
massfactor = ("critical", 200)

# Create NFW Object
n = NFW(mass=mass, concentration=concentration, redshift=redshift,
cosmo=cosmo,
        massfactor=massfactor)

# Radial distribution for plotting
radii = range(1,2001,10) * u.kpc

# Radial NFW density distribution
n_result = n(radii)

# Plot creation
fig, ax = plt.subplots(2)
fig.suptitle('1 Dimensional NFW Profile')
```

```
# Density profile subplot
ax[0].plot(radii, n_result, '-')
ax[0].set_yscale('log')
ax[0].set_xlabel(r"$r$ [{}]".format(radii.unit))
ax[0].set_ylabel(r"$\rho$ [{}]".format(n_result.unit))

# Create scaled density / scaled radius subplot
# NFW Object
n = NFW(mass=mass, concentration=concentration, redshift=redshift,
cosmo=cosmo,
        massfactor=massfactor)

# Radial distribution for plotting
radii = np.logspace(np.log10(1e-5), np.log10(2), num=1000) * u.Mpc
n_result = n(radii)

# Scaled density / scaled radius subplot
ax[1].plot(radii / n.radius_s, n_result / n.density_s, '-')
ax[1].set_xscale('log')
ax[1].set_yscale('log')
ax[1].set_xlabel(r"$r / r_s$")
ax[1].set_ylabel(r"$\rho / \rho_s$")

# Display plot
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

(png, svg, pdf)



The **`circular_velocity()`** member provides the circular velocity at each position `r` via the equation:

\[v_{circ}(r)^2=\frac{1}{x}\frac{\ln(1+cx)-(cx)/(1+cx)}{\ln(1+c)-c/(1+c)}\]

where x is the ratio `r` \(/r_{vir}\). Circular velocities are provided in km/s.

A sample plot of circular velocities of an NFW profile with the following parameters is displayed below:

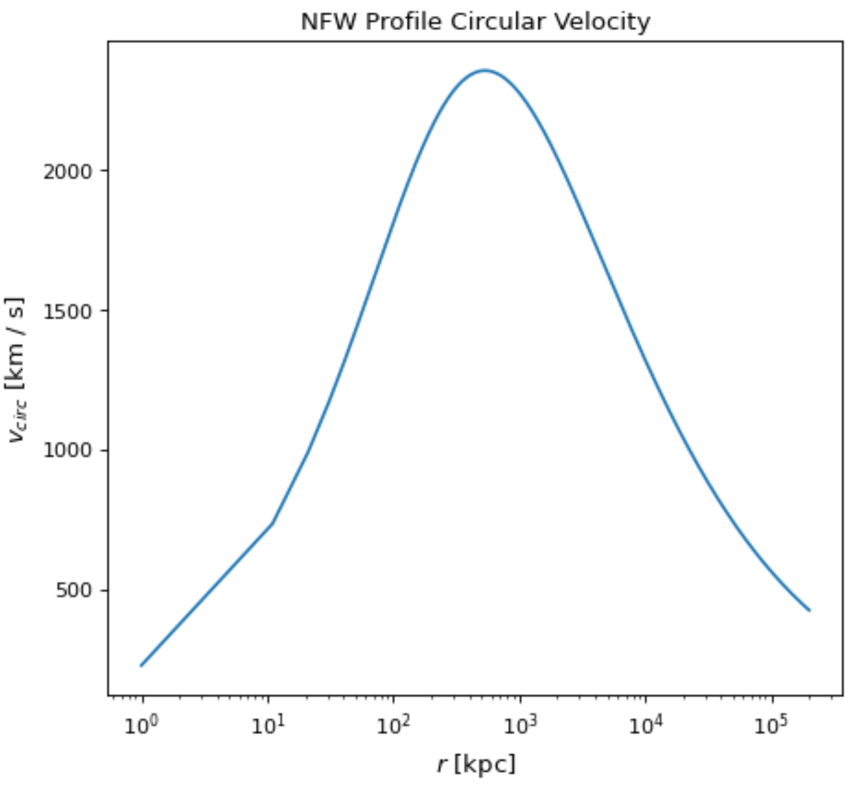
> `mass` = \(2.0 x 10^{15} M_{sun}\)
>
> `concentration` = 8.5
>
> `redshift` = 0.63

The maximum circular velocity and radius of maximum circular velocity are available as attributes `v_max` and `r_max`.

```
import matplotlib.pyplot as plt
from astropy.modeling.models import NFW
import astropy.units as u
from astropy import cosmology

# NFW Parameters
mass = u.Quantity(2.0E15, u.M_sun)
concentration = 8.5
```

```
redshift = 0.63
cosmo = cosmology.Planck15
massfactor = ("critical", 200)

# Create NFW Object
n = NFW(mass=mass, concentration=concentration, redshift=redshift,
cosmo=cosmo,
        massfactor=massfactor)

# Radial distribution for plotting
radii = range(1,200001,10) * u.kpc

# NFW circular velocity distribution
n_result = n.circular_velocity(radii)

# Plot creation
fig,ax = plt.subplots()
ax.set_title('NFW Profile Circular Velocity')
ax.plot(radii, n_result, '-')
ax.set_xscale('log')
ax.set_xlabel(r"$r$ [{}]".format(radii.unit))
ax.set_ylabel(r"$v_{circ}$" + " [{}]".format(n_result.unit))

# Display plot
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

(png, svg, pdf)



Polynomial Models

Notes regarding usage of domain and window

Most of the polynomial models have optional domain and window attributes. It is important to understand how they currently are interpreted, which can be confusing since the terminology often implies something different.

Both the domain and window attributes for a polynomial consist of a two element list (this will change to tuples in a future release) that indicate a range of values for input values. For 2-Dimensional polynomials the attributes become x_domain, y_domain, x_window, and y_window. Generally speaking, the main purpose of these attributes is to define a linear transform between the supplied input variable and the resultant input variable that is supplied to the polynomial. For example, if domain = [-2, 2] and window = [-1, 1], input values will be divided by two so that the domain maps to the window. Correspondingly the pair domain = [0, 2], window = [-1, 1] implies that 1 will be subtracted from the input variable before using it in the polynomial.

Neither domain or window are meant to imply that values that fall outside of their corresponding ranges will result in an exception, or that such values are necessarily invalid (the latter depends on the context of how the polynomial is being used).

It is the case that the orthogonal polynomials are defined on a range of [-1, 1], but nothing in the current machinery prevents them from being evaluated outside that range.

Domain is used in fitting polynomials to bound the input variable to map to the defined window so that they fall within the expected [-1, 1] range for such polynomials. That is, the fitting routine will set the domain to map to the window range for the range of input x values supplied (so that domain may change if the minimum and maximum x values being fit change).

The meaning of these terms may conflict with expectations (e.g., domain is often meant to mean the range of input values the funciton is valid for). For fit results that is somewhat true, but otherwise, it is not. The default values for ordinary polynomials is [-1, 1] for both domain and window, which effectively signals no transformation of the input variable.

The terminology was adopted from numpy polynomials, which have the same confusion in meaning.

1D Polynomials

- **`Polynomial1D`**
- **`Chebyshev1D`**
- **`Legendre1D`**
- **`Hermite1D`**

2D Polynomials

- **`Polynomial2D`**
- **`Chebyshev2D`**
- **`Legendre2D`**
- **`Hermite2D`**
- **`SIP`** model implements the Simple Imaging Polynomial (SIP) convention

Examples

Fitting a Line

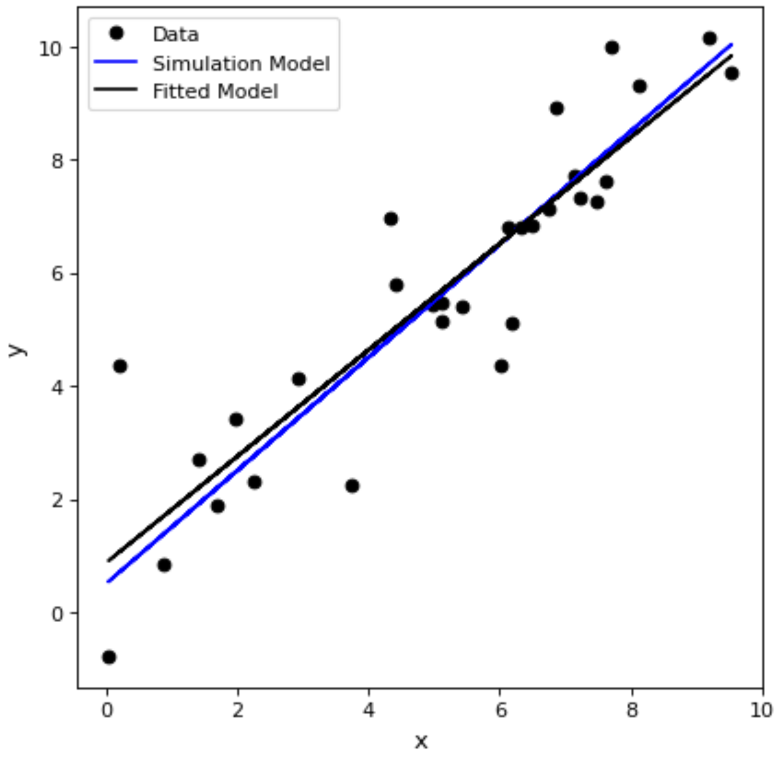
Fitting a line to (x,y) data points is a common case in many areas. Examples fits are given for fitting, fitting using the uncertainties as weights, and fitting using iterative sigma clipping.

Simple Fit

Here the (x,y) data points are fit with a line. The (x,y) data points are simulated and have a range of uncertainties to give a realistic example.

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.modeling import models, fitting

# define a model for a line
line_orig = models.Linear1D(slope=1.0, intercept=0.5)

# generate x, y data non-uniformly spaced in x
# add noise to y measurements
npts = 30
np.random.seed(10)
x = np.random.uniform(0.0, 10.0, npts)
y = line_orig(x)
yunc = np.absolute(np.random.normal(0.5, 2.5, npts))
y += np.random.normal(0.0, yunc, npts)

# initialize a linear fitter
fit = fitting.LinearLSQFitter()

# initialize a linear model
line_init = models.Linear1D()

# fit the data with the fitter
fitted_line = fit(line_init, x, y)

# plot
plt.figure()
plt.plot(x, y, 'ko', label='Data')
plt.plot(x, line_orig(x), 'b-', label='Simulation Model')
plt.plot(x, fitted_line(x), 'k-', label='Fitted Model')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
```

(png, svg, pdf)



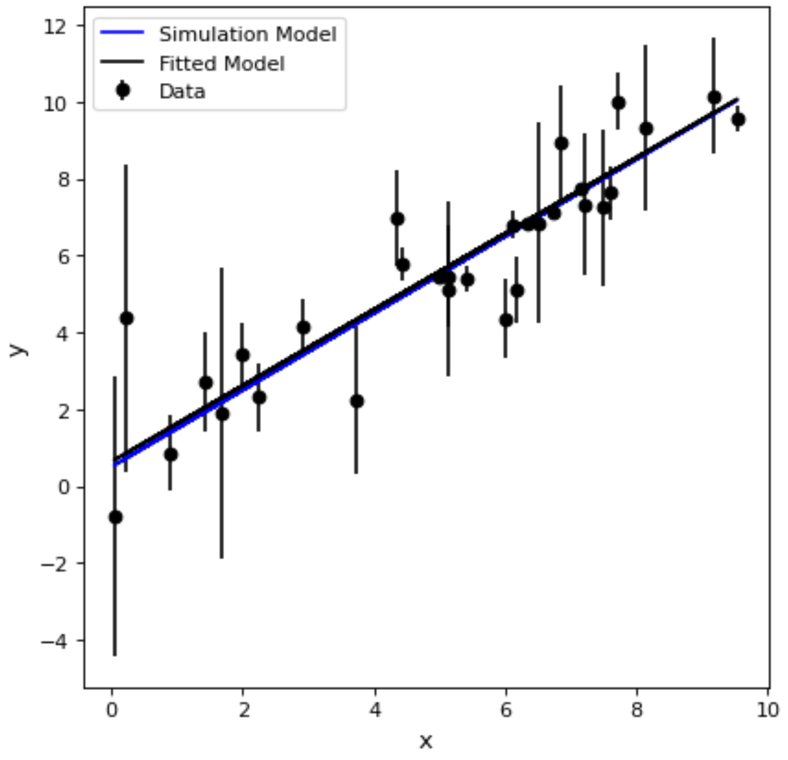
Fit using uncertainties

Fitting can be done using the uncertainties as weights. To get the standard weighting of 1/unc^2 for the case of Gaussian errors, the weights to pass to the fitting are 1/unc.

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.modeling import models, fitting

# define a model for a line
line_orig = models.Linear1D(slope=1.0, intercept=0.5)

# generate x, y data non-uniformly spaced in x
# add noise to y measurements
npts = 30
np.random.seed(10)
x = np.random.uniform(0.0, 10.0, npts)
y = line_orig(x)
yunc = np.absolute(np.random.normal(0.5, 2.5, npts))
y += np.random.normal(0.0, yunc, npts)

# initialize a linear fitter
fit = fitting.LinearLSQFitter()
```

```
# initialize a linear model
line_init = models.Linear1D()

# fit the data with the fitter
fitted_line = fit(line_init, x, y, weights=1.0/yunc)

# plot
plt.figure()
plt.errorbar(x, y, yerr=yunc, fmt='ko', label='Data')
plt.plot(x, line_orig(x), 'b-', label='Simulation Model')
plt.plot(x, fitted_line(x), 'k-', label='Fitted Model')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
```

(png, svg, pdf)



Iterative fitting using sigma clipping

When fitting, there may be data that are outliers from the fit that can significantly bias the fitting. These outliers can be identified and removed from the fitting iteratively. Note that the iterative sigma clipping assumes all the data have the same uncertainties for the sigma clipping decision.

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.stats import sigma_clip
from astropy.modeling import models, fitting

# define a model for a line
line_orig = models.Linear1D(slope=1.0, intercept=0.5)

# generate x, y data non-uniformly spaced in x
# add noise to y measurements
npts = 30
np.random.seed(10)
x = np.random.uniform(0.0, 10.0, npts)
y = line_orig(x)
yunc = np.absolute(np.random.normal(0.5, 2.5, npts))
y += np.random.normal(0.0, yunc, npts)

# make true outliers
y[3] = line_orig(x[3]) + 6 * yunc[3]
y[10] = line_orig(x[10]) - 4 * yunc[10]

# initialize a linear fitter
fit = fitting.LinearLSQFitter()

# initialize the outlier removal fitter
or_fit = fitting.FittingWithOutlierRemoval(fit, sigma_clip, niter=3,
sigma=3.0)

# initialize a linear model
line_init = models.Linear1D()

# fit the data with the fitter
fitted_line, mask = or_fit(line_init, x, y, weights=1.0/yunc)
filtered_data = np.ma.masked_array(y, mask=mask)

# plot
plt.figure()
plt.errorbar(x, y, yerr=yunc, fmt="ko", fillstyle="none",
label="Clipped Data")
plt.plot(x, filtered_data, "ko", label="Fitted Data")
plt.plot(x, line_orig(x), 'b-', label='Simulation Model')
plt.plot(x, fitted_line(x), 'k-', label='Fitted Model')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
```

Fitting with constraints

fitting support constraints, however, different fitters support different types of constraints. The **supported_constraints** attribute shows the type of constraints supported by a specific fitter:

```
>>> from astropy.modeling import fitting
>>> fitting.LinearLSQFitter.supported_constraints
['fixed']
>>> fitting.LevMarLSQFitter.supported_constraints
['fixed', 'tied', 'bounds']
>>> fitting.SLSQPLSQFitter.supported_constraints
['bounds', 'eqcons', 'ineqcons', 'fixed', 'tied']
```

Fixed Parameter Constraint

All fitters support fixed (frozen) parameters through the `fixed` argument to models or setting the **fixed** attribute directly on a parameter.

For linear fitters, freezing a polynomial coefficient means that the corresponding term will be subtracted from the data before fitting a polynomial without that term to the result. For example, fixing `c0` in a polynomial model will fit a polynomial with the zero-th order term missing to the data minus that constant. The fixed coefficients and corresponding terms are restored to the fit polynomial and this is the polynomial returned from the fitter:

```
  >>> import numpy as np
  >>> np.random.seed(seed=12345)
  >>> from astropy.modeling import models, fitting
  >>> x = np.arange(1, 10, .1)
  >>> p1 = models.Polynomial1D(2, c0=[1, 1], c1=[2, 2], c2=[3, 3],
  ...                          n_models=2)
  >>> p1  # doctest: +FLOAT_CMP
  <Polynomial1D(2, c0=[1., 1.], c1=[2., 2.], c2=[3., 3.],
n_models=2)>
  >>> y = p1(x, model_set_axis=False)
  >>> n = (np.random.randn(y.size)).reshape(y.shape)
  >>> p1.c0.fixed = True
  >>> pfit = fitting.LinearLSQFitter()
  >>> new_model = pfit(p1, x, y + n)  # doctest: +IGNORE_WARNINGS
  >>> print(new_model)  # doctest: +SKIP
  Model: Polynomial1D
  Inputs: ('x',)
  Outputs: ('y',)
  Model set size: 2
  Degree: 2
  Parameters:
       c0         c1                  c2
      --- ------------------ ------------------
      1.0  2.072116176718454   2.99115839177437
      1.0 1.9818866652726403 3.0024208951927585


The syntax to fix the same parameter ``c0`` using an argument to the
model
instead of ``p1.c0.fixed = True`` would be::

  >>> p1 = models.Polynomial1D(2, c0=[1, 1], c1=[2, 2], c2=[3, 3],
  ...                          n_models=2, fixed={'c0': True})
```

Bounded Constraints

Bounded fitting is supported through the `bounds` arguments to models or by setting **`min`** and **`max`** attributes on a parameter. Bounds for the **`LevMarLSQFitter`** are always exactly satisfied–if the value of the parameter is outside the fitting interval, it will be reset to the value at the bounds. The **`SLSQPLSQFitter`** optimization algorithm handles bounds internally.

Tied Constraints

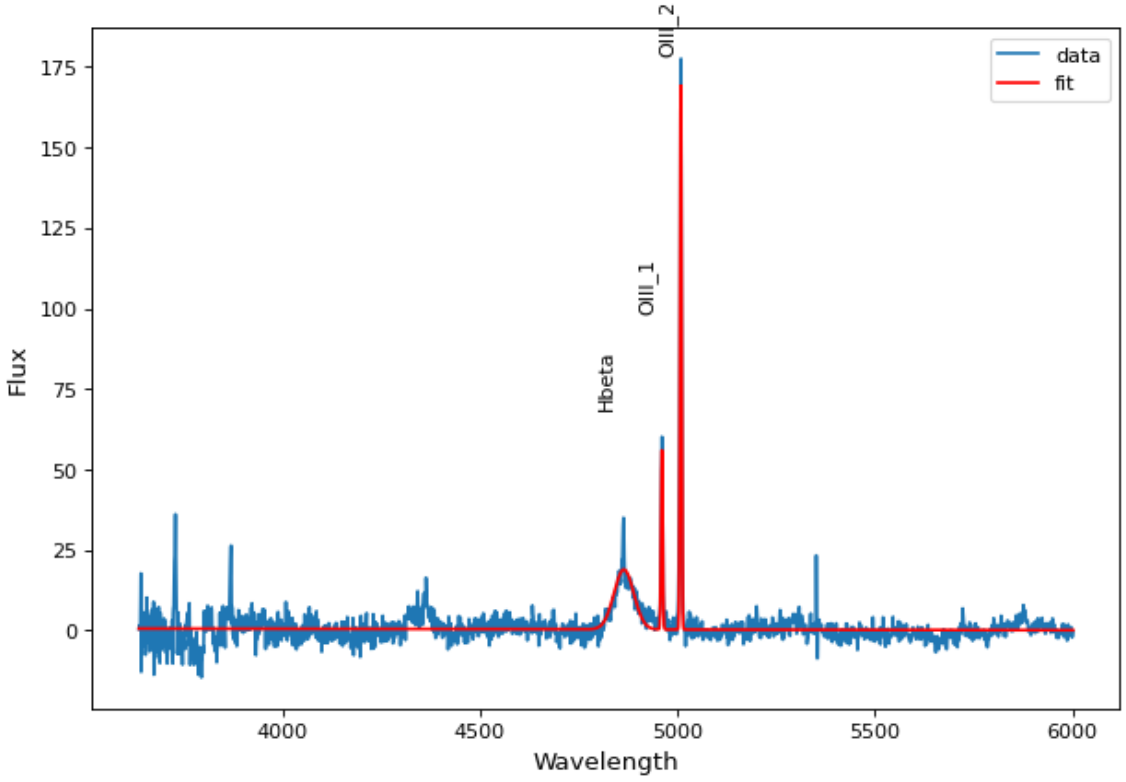
The **`tied`** constraint is often useful with Compound models. In this example we will read a spectrum from a file called `spec.txt` and fit Gaussians to the lines simultaneously while linking the flux of the OIII_1 and OIII_2 lines.

```
import numpy as np
from astropy.io import ascii
from astropy.utils.data import get_pkg_data_filename
from astropy.modeling import models, fitting
fname = get_pkg_data_filename('data/spec.txt',
package='astropy.modeling.tests')
spec = ascii.read(fname)
wave = spec['lambda']
flux = spec['flux']

# Use the rest wavelengths of known lines as initial values for the
fit.
```

```
Hbeta = 4862.721
OIII_1 = 4958.911
OIII_2 = 5008.239

# Create Gaussian1D models for each of the Hbeta and OIII lines.

h_beta = models.Gaussian1D(amplitude=34, mean=Hbeta, stddev=5)
o3_2 = models.Gaussian1D(amplitude=170, mean=OIII_2, stddev=5)
o3_1 = models.Gaussian1D(amplitude=57, mean=OIII_1, stddev=5)


# Tie the ratio of the intensity of the two OIII lines.

def tie_ampl(model):
    return model.amplitude_2 / 3.1

o3_1.amplitude.tied = tie_ampl


# Also tie the wavelength of the Hbeta line to the OIII wavelength.

def tie_wave(model):
    return model.mean_0 * OIII_1 / Hbeta

o3_1.mean.tied = tie_wave

# Create a Polynomial model to fit the continuum.

mean_flux = flux.mean()
cont = np.where(flux > mean_flux, mean_flux, flux)
linfitter = fitting.LinearLSQFitter()
poly_cont = linfitter(models.Polynomial1D(1), wave, cont)

# Create a compound model for the three lines and the continuum.

hbeta_combo = h_beta + o3_1 + o3_2 + poly_cont

# Fit all lines simultaneously.

fitter = fitting.LevMarLSQFitter()
fitted_model = fitter(hbeta_combo, wave, flux)
fitted_lines = fitted_model(wave)

from matplotlib import pyplot as plt
fig = plt.figure(figsize=(9, 6))
p = plt.plot(wave, flux, label="data")
p = plt.plot(wave, fitted_lines, 'r', label="fit")
p = plt.legend()
p = plt.xlabel("Wavelength")
```

```
p = plt.ylabel("Flux")
t = plt.text(4800, 70, 'Hbeta', rotation=90)
t = plt.text(4900, 100, 'OIII_1', rotation=90)
t = plt.text(4950, 180, 'OIII_2', rotation=90)
plt.show()
```

(png, svg, pdf)



Fitting Model Sets

Astropy model sets let you fit the same (linear) model to lots of independent data sets. It solves the linear equations simultaneously, so can avoid looping. But getting the data into the right shape can be a bit tricky.

The time savings could be worth the effort. In the example below, if we change the width*height of the data cube to 500*500 it takes 140 ms on a 2015 MacBook Pro to fit the models using model sets. Doing the same fit by looping over the 500*500 models takes 1.5 minutes, more than 600 times slower.

In the example below, we create a 3D data cube where the first dimension is a ramp – for example as from non-destructive readouts of an IR detector. So each pixel has a depth along a time axis, and flux that results a total number of counts that is increasing with time. We will be fitting a 1D polynomial vs. time to estimate the flux in counts/second (the slope of the fit). We will use just a small image of 3 rows by 4 columns, with a depth of 10 non-destructive reads.

First, import the necessary libraries:

```
>>> import numpy as np
>>> np.random.seed(seed=12345)
>>> from astropy.modeling import models, fitting
```

```
>>> depth, width, height = 10, 3, 4  # Time is along the depth axis
>>> t = np.arange(depth, dtype=np.float64)*10.  # e.g. readouts every
10 seconds
```

The number of counts in neach pixel is flux*time with the addition of some Gaussian noise:

```
>>> fluxes = np.arange(1. * width * height).reshape(width, height)
>>> image = fluxes[np.newaxis, :, :] * t[:, np.newaxis, np.newaxis]
>>> image += np.random.normal(0., image*0.05, size=image.shape)  #
Add noise
>>> image.shape
(10, 3, 4)
```

Create the models and the fitter. We need N=width*height instances of the same linear, parametric model (model sets currently only work with linear models and fitters):

```
>>> N = width * height
>>> line = models.Polynomial1D(degree=1, n_models=N)
>>> fit = fitting.LinearLSQFitter()
>>> print(f"We created {len(line)} models")
We created 12 models
```

We need to get the data to be fit into the right shape. It's not possible to just feed the 3D data cube. In this case, the time axis can be one dimensional. The fluxes have to be organized into an array that is of shape `width*height,depth` – in other words, we are reshaping to flatten last two axes and transposing to put them first:

```
>>> pixels = image.reshape((depth, width*height))
>>> y = pixels.T
>>> print("x axis is one dimensional: ",t.shape)
x axis is one dimensional:  (10,)
>>> print("y axis is two dimensional, N by len(x): ", y.shape)
y axis is two dimensional, N by len(x):  (12, 10)
```

Fit the model. It fits the N models simultaneously:

```
>>> new_model = fit(line, x=t, y=y)
>>> print(f"We fit {len(new_model)} models")
```

```
We fit 12 models
```

Fill an array with values computed from the best fit and reshape it to match the original:

```
>>> best_fit = new_model(t, model_set_axis=False).T.reshape((depth,
height, width))
>>> print("We reshaped the best fit to dimensions: ", best_fit.shape)
We reshaped the best fit to dimensions:  (10, 4, 3)
```

Now inspect the model:

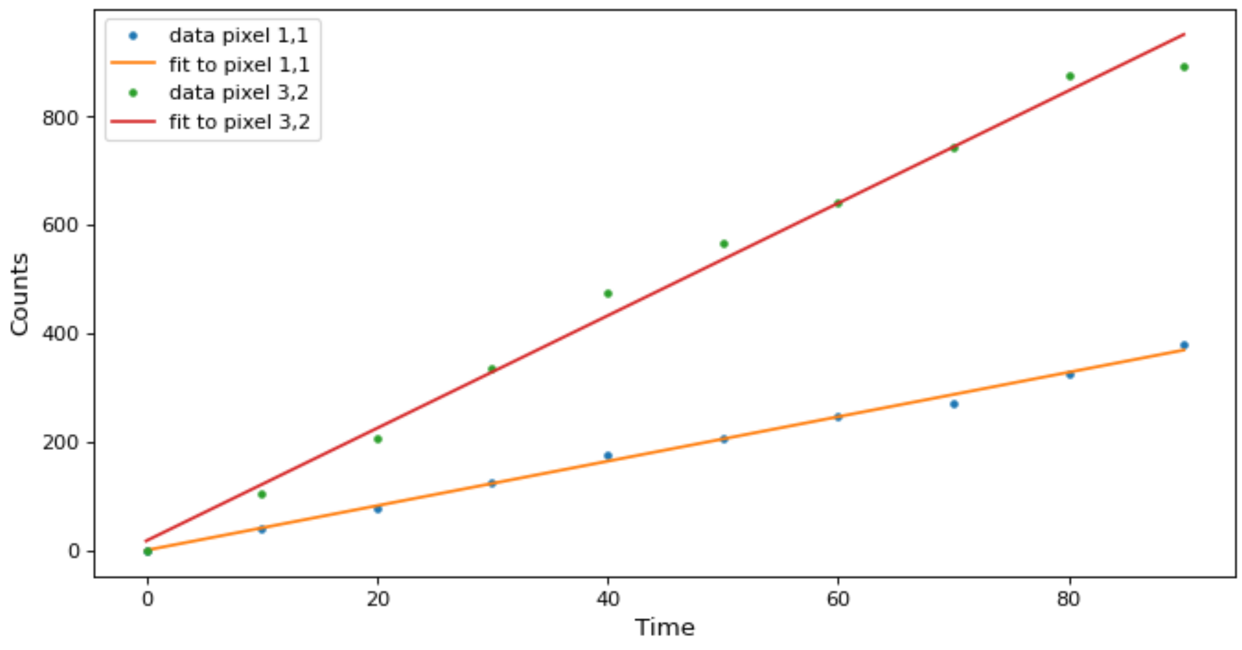
```
>>> print(new_model)
Model: Polynomial1D
Inputs: ('x',)
Outputs: ('y',)
Model set size: 12
Degree: 1
Parameters:
             c0                 c1
    ------------------- ------------------
                    0.0                0.0
    -0.5206606340901005 1.0463998276552442
     0.6401930368329991 1.9818733492667582
     0.1134712985541639  3.049279878262541
    -3.3556420351251313  4.013810434122983
      6.782223372575449  4.755912707001437
      3.628220497058842  5.841397947835126
    -5.8828309622531565  7.016044775363114
    -11.676538736037775  8.072519832452022
      -6.17932185981594  9.103924115403503
    -4.7258541419613165 10.315295021908833
       4.95631951675311 10.911167956770575

>>> print("The new_model has a param_sets attribute with shape:
",new_model.param_sets.shape)
The new_model has a param_sets attribute with shape:  (2, 12)

>>> print(f"And values that are the best-fit parameters for each
pixel:\n{new_model.param_sets}")
And values that are the best-fit parameters for each pixel:
[[  0.          -0.52066063   0.64019304   0.1134713   -3.35564204
    6.78222337   3.6282205   -5.88283096 -11.67653874  -6.17932186
   -4.72585414   4.95631952]
 [  0.           1.04639983   1.98187335   3.04927988   4.01381043
    4.75591271   5.84139795   7.01604478   8.07251983   9.10392412
   10.31529502  10.91116796]]
```

Plot the fit along a couple of pixels:

```
>>> def plotramp(t, image, best_fit, row, col):
...     plt.plot(t, image[:, row, col], '.', label=f'data pixel {row},{col}')
...     plt.plot(t, best_fit[:, row, col], '-', label=f'fit to pixel {row},{col}')
...     plt.xlabel('Time')
...     plt.ylabel('Counts')
...     plt.legend(loc='upper left')
>>> fig = plt.figure(figsize=(10, 5))
>>> plotramp(t, image, best_fit, 1, 1)
>>> plotramp(t, image, best_fit, 2, 1)
```

The data and the best fit model are shown together on one plot.

(png, svg, pdf)



Reference/API

Reference/API

Capabilities

astropy.modeling Package

This subpackage provides a framework for representing models and performing model evaluation and fitting. It supports 1D and 2D models and fitting with parameter constraints. It has some predefined models and fitting routines.

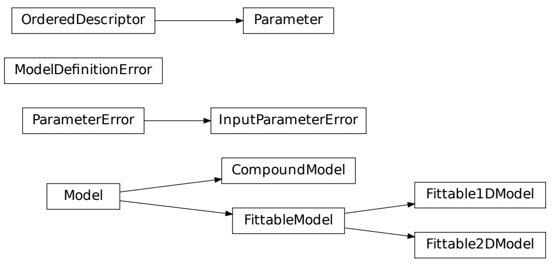
Functions

`custom_model`(*args[, fit_deriv])	Create a model from a user defined function.
`fix_inputs`(modelinstance, values)	This function creates a compound model with one or more of the input values of the input model assigned fixed values (scalar or array).
`is_separable`(transform)	A separability test for the outputs of a transform.
`separability_matrix`(transform)	Compute the correlation between outputs and inputs.

Classes

`CompoundModel`(op, left, right[, name, inverse])	Base class for compound models.
`Fittable1DModel`(*args[, meta, name])	Base class for one-dimensional fittable models.
`Fittable2DModel`(*args[, meta, name])	Base class for two-dimensional fittable models.
`FittableModel`(*args[, meta, name])	Base class for models that can be fitted using the built-in fitting algorithms.
`InputParameterError`	Used for incorrect input parameter values and definitions.
`Model`(*args[, meta, name])	Base class for all models.
`ModelDefinitionError`	Used for incorrect models definitions.
`Parameter`([name, description, default, …])	Wraps individual parameters.
`ParameterError`	Generic exception class for all exceptions pertaining to Parameters.

Class Inheritance Diagram



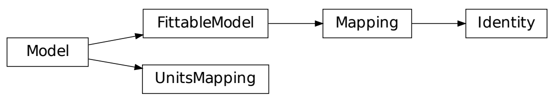
astropy.modeling.mappings Module

Special models useful for complex compound models where control is needed over which outputs from a source model are mapped to which inputs of a target model.

Classes

`Mapping`(mapping[, n_inputs, name, meta])	Allows inputs to be reordered, duplicated or dropped.
`Identity`(n_inputs[, name, meta])	Returns inputs unchanged.
`UnitsMapping`(mapping[, …])	Mapper that operates on the units of the input, first converting to canonical units, then assigning new units without further conversion.

Class Inheritance Diagram



astropy.modeling.fitting Module

This module implements classes (called Fitters) which combine optimization algorithms (typically from `scipy.optimize`) with statistic functions to perform fitting. Fitters are implemented as callable classes. In addition to the data to fit, the `__call__` method takes an instance of `FittableModel` as input, and returns a copy of the model with its parameters determined by the optimizer.

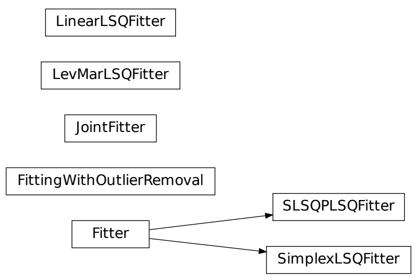
Optimization algorithms, called “optimizers” are implemented in `optimizers` and statistic functions are in `statistic`. The goal is to provide an easy to extend framework and allow users to easily create new fitters by combining statistics with optimizers.

There are two exceptions to the above scheme. `LinearLSQFitter` uses Numpy’s `lstsq` function. `LevMarLSQFitter` uses `leastsq` which combines optimization and statistic in one implementation.

Classes

`LinearLSQFitter`([calc_uncertainties])	A class performing a linear least square fitting.
`LevMarLSQFitter`([calc_uncertainties])	Levenberg-Marquardt algorithm and least squares statistic.
`FittingWithOutlierRemoval`(fitter, outlier_func)	This class combines an outlier removal technique with a fitting procedure.
`SLSQPLSQFitter`()	Sequential Least Squares Programming (SLSQP) optimization algorithm and least squares statistic.
`SimplexLSQFitter`()	Simplex algorithm and least squares statistic.
`JointFitter`(models, jointparameters, initvals)	Fit models which share a parameter.
`Fitter`(optimizer, statistic)	Base class for all fitters.

Class Inheritance Diagram



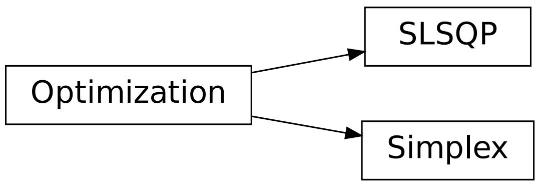
astropy.modeling.optimizers Module

Optimization algorithms used in `fitting`.

Classes

`Optimization`(opt_method)	Base class for optimizers.
`SLSQP`()	Sequential Least Squares Programming optimization algorithm.
`Simplex`()	Neald-Mead (downhill simplex) algorithm.

Class Inheritance Diagram



astropy.modeling.statistic Module

Statistic functions used in `fitting`.

Functions

`leastsquare`(measured_vals, updated_model, …)	Least square statistic, with optional weights, in N-dimensions.
`leastsquare_1d`(measured_vals, updated_model, …)	Least square statistic with optional weights.
`leastsquare_2d`(measured_vals, updated_model, …)	Least square statistic with optional weights.
`leastsquare_3d`(measured_vals, updated_model, …)	Least square statistic with optional weights.

astropy.modeling.separable Module

Functions to determine if a model is separable, i.e. if the model outputs are independent.

It analyzes `n_inputs`, `n_outputs` and the operators in a compound model by stepping through the transforms and creating a `coord_matrix` of shape (`n_outputs`, `n_inputs`).

Each modeling operator is represented by a function which takes two simple models (or two `coord_matrix` arrays) and returns an array of shape (`n_outputs`, `n_inputs`).

Functions

`is_separable`(transform)	A separability test for the outputs of a transform.
`separability_matrix`(transform)	Compute the correlation between outputs and inputs.

Pre-Defined Models

astropy.modeling.functional_models Module

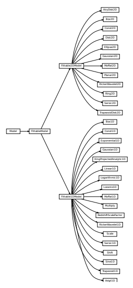
Mathematical models.

Classes

`AiryDisk2D`([amplitude, x_0, y_0, radius])	Two dimensional Airy disk model.
`Moffat1D`([amplitude, x_0, gamma, alpha])	One dimensional Moffat model.
`Moffat2D`([amplitude, x_0, y_0, gamma, alpha])	Two dimensional Moffat model.
`Box1D`([amplitude, x_0, width])	One dimensional Box model.
`Box2D`([amplitude, x_0, y_0, x_width, y_width])	Two dimensional Box model.
`Const1D`([amplitude])	One dimensional Constant model.
`Const2D`([amplitude])	Two dimensional Constant model.
`Ellipse2D`([amplitude, x_0, y_0, a, b, theta])	A 2D Ellipse model.
`Disk2D`([amplitude, x_0, y_0, R_0])	Two dimensional radial symmetric Disk model.
`Gaussian1D`([amplitude, mean, stddev])	One dimensional Gaussian model.
`Gaussian2D`([amplitude, x_mean, y_mean, …])	Two dimensional Gaussian model.
`Linear1D`([slope, intercept])	One dimensional Line model.
`Lorentz1D`([amplitude, x_0, fwhm])	One dimensional Lorentzian model.
`RickerWavelet1D`([amplitude, x_0, sigma])	One dimensional Ricker Wavelet model (sometimes known as a “Mexican Hat” model).
`RickerWavelet2D`([amplitude, x_0, y_0, sigma])	Two dimensional Ricker Wavelet model (sometimes known as a “Mexican Hat” model).
`RedshiftScaleFactor`([z])	One dimensional redshift scale factor model.
`Multiply`([factor])	Multiply a model by a quantity or number.
`Planar2D`([slope_x, slope_y, intercept])	Two dimensional Plane model.
`Scale`([factor])	Multiply a model by a dimensionless factor.
`Sersic1D`([amplitude, r_eff, n])	One dimensional Sersic surface brightness profile.
`Sersic2D`([amplitude, r_eff, n, x_0, y_0, …])	Two dimensional Sersic surface brightness profile.
`Shift`([offset])	Shift a coordinate.
`Sine1D`([amplitude, frequency, phase])	One dimensional Sine model.
`Trapezoid1D`([amplitude, x_0, width, slope])	One dimensional Trapezoid model.
`TrapezoidDisk2D`([amplitude, x_0, y_0, R_0, …])	Two dimensional circular Trapezoid model.
`Ring2D`([amplitude, x_0, y_0, r_in, width, r_out])	Two dimensional radial symmetric Ring model.

Voigt1D([x_0, amplitude_L, fwhm_L, fwhm_G])	One dimensional model for the Voigt profile.
KingProjectedAnalytic1D([amplitude, r_core, …])	Projected (surface density) analytic King Model.
Exponential1D([amplitude, tau])	One dimensional exponential model.
Logarithmic1D([amplitude, tau])	One dimensional logarithmic model.

Class Inheritance Diagram



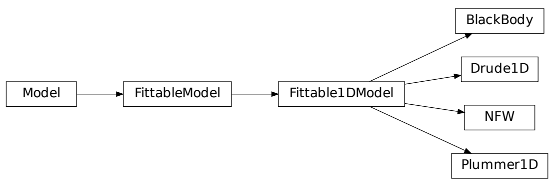
astropy.modeling.physical_models Module

Models that have physical origins.

Classes

BlackBody([temperature, scale])	Blackbody model using the Planck function.
Drude1D([amplitude, x_0, fwhm])	Drude model based one the behavior of electons in materials (esp.
Plummer1D([mass, r_plum])	One dimensional Plummer density profile model.
NFW([mass, concentration, redshift, …])	Navarro–Frenk–White (NFW) profile - model for radial distribution of dark matter.

Class Inheritance Diagram



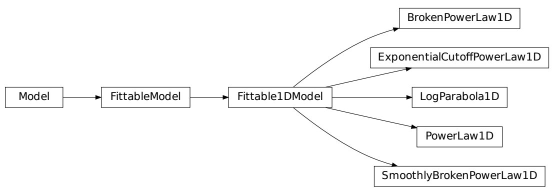
astropy.modeling.powerlaws Module

Power law model variants

Classes

PowerLaw1D([amplitude, x_0, alpha])	One dimensional power law model.
BrokenPowerLaw1D([amplitude, x_break, …])	One dimensional power law model with a break.
SmoothlyBrokenPowerLaw1D([amplitude, …])	One dimensional smoothly broken power law model.
ExponentialCutoffPowerLaw1D([amplitude, …])	One dimensional power law model with an exponential cutoff.
LogParabola1D([amplitude, x_0, alpha, beta])	One dimensional log parabola model (sometimes called curved power law).

Class Inheritance Diagram



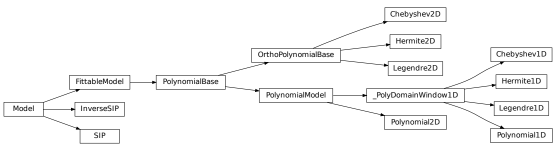
astropy.modeling.polynomial Module

This module contains models representing polynomials and polynomial series.

Classes

Chebyshev1D(degree[, domain, window, …])	Univariate Chebyshev series.
Chebyshev2D(x_degree, y_degree[, x_domain, …])	Bivariate Chebyshev series..
Hermite1D(degree[, domain, window, …])	Univariate Hermite series.
Hermite2D(x_degree, y_degree[, x_domain, …])	Bivariate Hermite series.
InverseSIP(ap_order, bp_order[, ap_coeff, …])	Inverse Simple Imaging Polynomial
Legendre1D(degree[, domain, window, …])	Univariate Legendre series.
Legendre2D(x_degree, y_degree[, x_domain, …])	Bivariate Legendre series.
Polynomial1D(degree[, domain, window, …])	1D Polynomial model.
Polynomial2D(degree[, x_domain, y_domain, …])	2D Polynomial model.
SIP(crpix, a_order, b_order[, a_coeff, …])	Simple Imaging Polynomial (SIP) model.
OrthoPolynomialBase(x_degree, y_degree[, …])	This is a base class for the 2D Chebyshev and Legendre models.
PolynomialModel(degree[, n_models, …])	Base class for polynomial models.

Class Inheritance Diagram



astropy.modeling.projections Module

Implements projections–particularly sky projections defined in WCS Paper II [1].

All angles are set and and displayed in degrees but internally computations are performed in radians. All functions expect inputs and outputs degrees.

References

[1] Calabretta, M.R., Greisen, E.W., 2002, A&A, 395, 1077 (Paper II)

Classes

Projection(*args[, meta, name])	Base class for all sky projections.
Pix2SkyProjection(*args, **kwargs)	Base class for all Pix2Sky projections.
Sky2PixProjection(*args, **kwargs)	Base class for all Sky2Pix projections.
Zenithal(*args[, meta, name])	Base class for all Zenithal projections.
Cylindrical(*args[, meta, name])	Base class for Cylindrical projections.
PseudoCylindrical(*args[, meta, name])	Base class for pseudocylindrical projections.
Conic(*args[, meta, name])	Base class for conic projections.
PseudoConic(*args[, meta, name])	Base class for pseudoconic projections.
QuadCube(*args[, meta, name])	Base class for quad cube projections.
HEALPix(*args[, meta, name])	Base class for HEALPix projections.
AffineTransformation2D([matrix, translation])	Perform an affine transformation in 2 dimensions.
Pix2Sky_ZenithalPerspective([mu, gamma])	Zenithal perspective projection - pixel to sky.
Sky2Pix_ZenithalPerspective([mu, gamma])	Zenithal perspective projection - sky to pixel.
Pix2Sky_SlantZenithalPerspective([mu, phi0, …])	Slant zenithal perspective projection - pixel to sky.
Sky2Pix_SlantZenithalPerspective([mu, phi0, …])	Zenithal perspective projection - sky to pixel.
Pix2Sky_Gnomonic(*args, **kwargs)	Gnomonic projection - pixel to sky.
Sky2Pix_Gnomonic(*args, **kwargs)	Gnomonic Projection - sky to pixel.
Pix2Sky_Stereographic(*args, **kwargs)	Stereographic Projection - pixel to sky.
Sky2Pix_Stereographic(*args, **kwargs)	Stereographic Projection - sky to pixel.
Pix2Sky_SlantOrthographic([xi, eta])	Slant orthographic projection - pixel to sky.
Sky2Pix_SlantOrthographic([xi, eta])	Slant orthographic projection - sky to pixel.
Pix2Sky_ZenithalEquidistant(*args, **kwargs)	Zenithal equidistant projection - pixel to sky.
Sky2Pix_ZenithalEquidistant(*args, **kwargs)	Zenithal equidistant projection - sky to pixel.
Pix2Sky_ZenithalEqualArea(*args, **kwargs)	Zenithal equidistant projection - pixel to sky.
Sky2Pix_ZenithalEqualArea(*args, **kwargs)	Zenithal equidistant projection - sky to pixel.
Pix2Sky_Airy([theta_b])	Airy projection - pixel to sky.
Sky2Pix_Airy([theta_b])	Airy - sky to pixel.
Pix2Sky_CylindricalPerspective([mu, lam])	Cylindrical perspective - pixel to sky.
Sky2Pix_CylindricalPerspective([mu, lam])	Cylindrical Perspective - sky to pixel.

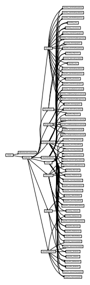
Pix2Sky_CylindricalEqualArea([lam])	Cylindrical equal area projection - pixel to sky.
Sky2Pix_CylindricalEqualArea([lam])	Cylindrical equal area projection - sky to pixel.
Pix2Sky_PlateCarree(*args, **kwargs)	Plate carrée projection - pixel to sky.
Sky2Pix_PlateCarree(*args, **kwargs)	Plate carrée projection - sky to pixel.
Pix2Sky_Mercator(*args, **kwargs)	Mercator - pixel to sky.
Sky2Pix_Mercator(*args, **kwargs)	Mercator - sky to pixel.
Pix2Sky_SansonFlamsteed(*args, **kwargs)	Sanson-Flamsteed projection - pixel to sky.
Sky2Pix_SansonFlamsteed(*args, **kwargs)	Sanson-Flamsteed projection - sky to pixel.
Pix2Sky_Parabolic(*args, **kwargs)	Parabolic projection - pixel to sky.
Sky2Pix_Parabolic(*args, **kwargs)	Parabolic projection - sky to pixel.
Pix2Sky_Molleweide(*args, **kwargs)	Molleweide's projection - pixel to sky.
Sky2Pix_Molleweide(*args, **kwargs)	Molleweide's projection - sky to pixel.
Pix2Sky_HammerAitoff(*args, **kwargs)	Hammer-Aitoff projection - pixel to sky.
Sky2Pix_HammerAitoff(*args, **kwargs)	Hammer-Aitoff projection - sky to pixel.
Pix2Sky_ConicPerspective(*args, **kwargs)	Colles' conic perspective projection - pixel to sky.
Sky2Pix_ConicPerspective(*args, **kwargs)	Colles' conic perspective projection - sky to pixel.
Pix2Sky_ConicEqualArea(*args, **kwargs)	Alber's conic equal area projection - pixel to sky.
Sky2Pix_ConicEqualArea(*args, **kwargs)	Alber's conic equal area projection - sky to pixel.
Pix2Sky_ConicEquidistant(*args, **kwargs)	Conic equidistant projection - pixel to sky.
Sky2Pix_ConicEquidistant(*args, **kwargs)	Conic equidistant projection - sky to pixel.
Pix2Sky_ConicOrthomorphic(*args, **kwargs)	Conic orthomorphic projection - pixel to sky.
Sky2Pix_ConicOrthomorphic(*args, **kwargs)	Conic orthomorphic projection - sky to pixel.
Pix2Sky_BonneEqualArea([theta1])	Bonne's equal area pseudoconic projection - pixel to sky.
Sky2Pix_BonneEqualArea([theta1])	Bonne's equal area pseudoconic projection - sky to pixel.
Pix2Sky_Polyconic(*args, **kwargs)	Polyconic projection - pixel to sky.
Sky2Pix_Polyconic(*args, **kwargs)	Polyconic projection - sky to pixel.
Pix2Sky_TangentialSphericalCube(*args, **kwargs)	Tangential spherical cube projection - pixel to sky.
Sky2Pix_TangentialSphericalCube(*args, **kwargs)	Tangential spherical cube projection - sky to pixel.
Pix2Sky_COBEQuadSphericalCube(*args, **kwargs)	COBE quadrilateralized spherical cube projection - pixel to sky.
Sky2Pix_COBEQuadSphericalCube(*args, **kwargs)	COBE quadrilateralized spherical cube projection - sky to pixel.
Pix2Sky_QuadSphericalCube(*args, **kwargs)	Quadrilateralized spherical cube projection - pixel to sky.

Sky2Pix_QuadSphericalCube(*args, **kwargs)	Quadrilateralized spherical cube projection - sky to pixel.
Pix2Sky_HEALPix([H, X])	HEALPix - pixel to sky.
Sky2Pix_HEALPix([H, X])	HEALPix projection - sky to pixel.
Pix2Sky_HEALPixPolar(*args, **kwargs)	HEALPix polar, aka “butterfly” projection - pixel to sky.
Sky2Pix_HEALPixPolar(*args, **kwargs)	HEALPix polar, aka “butterfly” projection - pixel to sky.
Pix2Sky_AZP	alias of astropy.modeling.projections.Pix2Sky_ZenithalPerspect
Sky2Pix_AZP	alias of astropy.modeling.projections.Sky2Pix_ZenithalPerspect
Pix2Sky_SZP	alias of astropy.modeling.projections.Pix2Sky_SlantZenithalPersp
Sky2Pix_SZP	alias of astropy.modeling.projections.Sky2Pix_SlantZenithalPersp
Pix2Sky_TAN	alias of astropy.modeling.projections.Pix2Sky_Gnomoni
Sky2Pix_TAN	alias of astropy.modeling.projections.Sky2Pix_Gnomoni
Pix2Sky_STG	alias of astropy.modeling.projections.Pix2Sky_Stereograp
Sky2Pix_STG	alias of astropy.modeling.projections.Sky2Pix_Stereograp
Pix2Sky_SIN	alias of astropy.modeling.projections.Pix2Sky_SlantOrthogr
Sky2Pix_SIN	alias of astropy.modeling.projections.Sky2Pix_SlantOrthogr
Pix2Sky_ARC	alias of astropy.modeling.projections.Pix2Sky_ZenithalEquidist
Sky2Pix_ARC	alias of astropy.modeling.projections.Sky2Pix_ZenithalEquidist
Pix2Sky_ZEA	alias of astropy.modeling.projections.Pix2Sky_ZenithalEqua
Sky2Pix_ZEA	alias of astropy.modeling.projections.Sky2Pix_ZenithalEqua
Pix2Sky_AIR	alias of astropy.modeling.projections.Pix2Sky_Airy
Sky2Pix_AIR	alias of astropy.modeling.projections.Sky2Pix_Airy
Pix2Sky_CYP	alias of astropy.modeling.projections.Pix2Sky_CylindricalPerspe
Sky2Pix_CYP	alias of astropy.modeling.projections.Sky2Pix_CylindricalPerspe
Pix2Sky_CEA	alias of astropy.modeling.projections.Pix2Sky_CylindricalEqual

Sky2Pix_CEA	alias of astropy.modeling.projections.Sky2Pix_CylindricalEqual
Pix2Sky_CAR	alias of astropy.modeling.projections.Pix2Sky_PlateCarr
Sky2Pix_CAR	alias of astropy.modeling.projections.Sky2Pix_PlateCarr
Pix2Sky_MER	alias of astropy.modeling.projections.Pix2Sky_Mercato
Sky2Pix_MER	alias of astropy.modeling.projections.Sky2Pix_Mercato
Pix2Sky_SFL	alias of astropy.modeling.projections.Pix2Sky_SansonFlams
Sky2Pix_SFL	alias of astropy.modeling.projections.Sky2Pix_SansonFlams
Pix2Sky_PAR	alias of astropy.modeling.projections.Pix2Sky_Paraboli
Sky2Pix_PAR	alias of astropy.modeling.projections.Sky2Pix_Paraboli
Pix2Sky_MOL	alias of astropy.modeling.projections.Pix2Sky_Mollwei
Sky2Pix_MOL	alias of astropy.modeling.projections.Sky2Pix_Mollwei
Pix2Sky_AIT	alias of astropy.modeling.projections.Pix2Sky_HammerAit
Sky2Pix_AIT	alias of astropy.modeling.projections.Sky2Pix_HammerAit
Pix2Sky_COP	alias of astropy.modeling.projections.Pix2Sky_ConicPerspe
Sky2Pix_COP	alias of astropy.modeling.projections.Sky2Pix_ConicPerspe
Pix2Sky_COE	alias of astropy.modeling.projections.Pix2Sky_ConicEqual
Sky2Pix_COE	alias of astropy.modeling.projections.Sky2Pix_ConicEqual
Pix2Sky_COD	alias of astropy.modeling.projections.Pix2Sky_ConicEquidi
Sky2Pix_COD	alias of astropy.modeling.projections.Sky2Pix_ConicEquidi
Pix2Sky_COO	alias of astropy.modeling.projections.Pix2Sky_ConicOrthomo
Sky2Pix_COO	alias of astropy.modeling.projections.Sky2Pix_ConicOrthomo
Pix2Sky_BON	alias of astropy.modeling.projections.Pix2Sky_BonneEqual
Sky2Pix_BON	alias of astropy.modeling.projections.Sky2Pix_BonneEqual
Pix2Sky_PCO	alias of astropy.modeling.projections.Pix2Sky_Polyconi
Sky2Pix_PCO	alias of astropy.modeling.projections.Sky2Pix_Polyconi
Pix2Sky_TSC	alias of astropy.modeling.projections.Pix2Sky_TangentialSpherica
Sky2Pix_TSC	alias of astropy.modeling.projections.Sky2Pix_TangentialSpherica
Pix2Sky_CSC	alias of astropy.modeling.projections.Pix2Sky_COBEQuadSpherical

Sky2Pix_CSC	alias of astropy.modeling.projections.Sky2Pix_COBEQuadSpherical
Pix2Sky_QSC	alias of astropy.modeling.projections.Pix2Sky_QuadSpherica
Sky2Pix_QSC	alias of astropy.modeling.projections.Sky2Pix_QuadSpherica
Pix2Sky_HPX	alias of astropy.modeling.projections.Pix2Sky_HEALPix
Sky2Pix_HPX	alias of astropy.modeling.projections.Sky2Pix_HEALPix
Pix2Sky_XPH	alias of astropy.modeling.projections.Pix2Sky_HEALPixPo
Sky2Pix_XPH	alias of astropy.modeling.projections.Sky2Pix_HEALPixPo

Class Inheritance Diagram



astropy.modeling.rotations Module

Implements rotations, including spherical rotations as defined in WCS Paper II [1]

`RotateNative2Celestial` and `RotateCelestial2Native` follow the convention in WCS Paper II to rotate to/from a native sphere and the celestial sphere.

The implementation uses `EulerAngleRotation`. The model parameters are three angles: the longitude (`lon`) and latitude (`lat`) of the fiducial point in the celestial system (`CRVAL` keywords in FITS), and the longitude of the celestial pole in the native system (`lon_pole`). The Euler angles are `lon+90` , `90-lat` and `-(lon_pole-90)` .

References

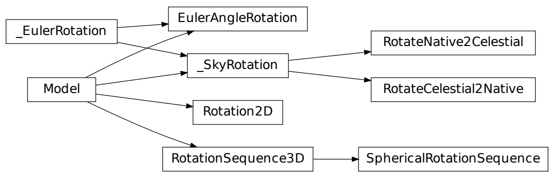
[1] Calabretta, M.R., Greisen, E.W., 2002, A&A, 395, 1077 (Paper II)

Classes

RotateCelestial2Native(lon, lat, lon_pole, …)	Transform from Celestial to Native Spherical Coordinates.
RotateNative2Celestial(lon, lat, lon_pole, …)	Transform from Native to Celestial Spherical Coordinates.
Rotation2D([angle])	Perform a 2D rotation given an angle.
EulerAngleRotation(phi, theta, psi, …)	Implements Euler angle intrinsic rotations.
RotationSequence3D(angles, axes_order[, name])	Perform a series of rotations about different axis in 3D space.

SphericalRotationSequence(angles, axes_order)	Perform a sequence of rotations about arbitrary number of axes in spherical coordinates.

Class Inheritance Diagram



astropy.modeling.tabular Module

Tabular models.

Tabular models of any dimension can be created using `tabular_model`. For convenience `Tabular1D` and `Tabular2D` are provided.

Examples

```
>>> table = np.array([[ 3.,  0.,  0.],
...                   [ 0.,  2.,  0.],
...                   [ 0.,  0.,  0.]])
>>> points = ([1, 2, 3], [1, 2, 3])
>>> t2 = Tabular2D(points, lookup_table=table, bounds_error=False,
...                fill_value=None, method='nearest')
```

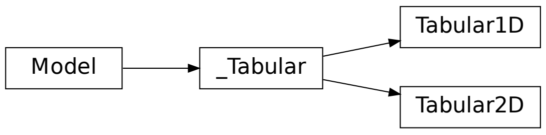
Functions

tabular_model(dim[, name])	Make a `Tabular` model where `n_inputs` is based on the dimension of the lookup_table.

Classes

Tabular1D([points, lookup_table, method, …])	Tabular model in 1D.
Tabular2D([points, lookup_table, method, …])	Tabular model in 2D.

Class Inheritance Diagram



Uncertainties and Distributions

(astropy.uncertainty)

Note

astropy.uncertainty is relatively new (`astropy` v3.1), and thus it is possible there will be API changes in upcoming versions of `astropy`. If you have specific ideas for how it might be improved, please let us know on the astropy-dev mailing list or at http://feedback.astropy.org.

Introduction

`astropy` provides a **Distribution** object to represent statistical distributions in a form that acts as a drop-in replacement for a **Quantity** object or a regular **numpy.ndarray**. Used in this manner, **Distribution** provides uncertainty propagation at the cost of additional computation. It can also more generally represent sampled distributions for Monte Carlo calculation techniques, for instance.

The core object for this feature is the **Distribution**. Currently, all such distributions are Monte Carlo sampled. While this means each distribution may take more memory, it allows arbitrarily complex operations to be performed on distributions while maintaining their correlation structure. Some specific well-behaved distributions (e.g., the normal distribution) have analytic forms which may eventually enable a more compact and efficient representation. In the future, these may provide a coherent uncertainty propagation mechanism to work with **NDData**. However, this is not currently implemented. Hence, details of storing uncertainties for **NDData** objects can be found in the N-Dimensional Datasets (astropy.nddata) section.

Getting Started

To demonstrate a basic use case for distributions, consider the problem of uncertainty propagation of normal distributions. Assume there are two measurements you wish to add, each with normal uncertainties. We start with some initial imports and setup:

```
>>> import numpy as np
>>> from astropy import units as u
>>> from astropy import uncertainty as unc
>>> np.random.seed(12345)  # ensures reproducible example numbers
```

Now we create two **Distribution** objects to represent our distributions:

```
>>> a = unc.normal(1*u.kpc, std=30*u.pc, n_samples=10000)
>>> b = unc.normal(2*u.kpc, std=40*u.pc, n_samples=10000)
```

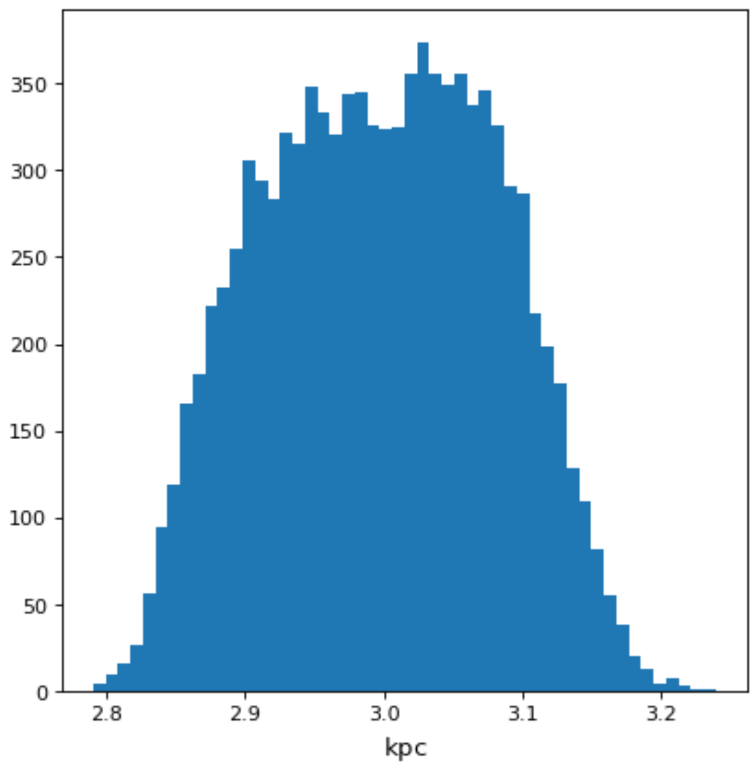
For normal distributions, the centers should add as expected, and the standard deviations add in quadrature. We can check these results (to the limits of our Monte Carlo sampling) trivially with **Distribution** arithmetic and attributes:

```
>>> c = a + b
>>> c
<QuantityDistribution [...] kpc with n_samples=10000>
>>> c.pdf_mean()
<Quantity 2.99970555 kpc>
>>> c.pdf_std().to(u.pc)
<Quantity 50.07120457 pc>
```

Indeed these are close to the expectations. While this may seem unnecessary for the basic Gaussian case, for more complex distributions or arithmetic operations where error analysis becomes untenable, **Distribution** still powers through:

```
>>> d = unc.uniform(center=3*u.kpc, width=800*u.pc, n_samples=10000)
>>> e = unc.Distribution(((np.random.beta(2,5, 10000)-(2/7))/2 +
3)*u.kpc)
>>> f = (c * d * e) ** (1/3)
>>> f.pdf_mean()
<Quantity 2.99786227 kpc>
>>> f.pdf_std()
<Quantity 0.08330476 kpc>
>>> from matplotlib import pyplot as plt
>>> from astropy.visualization import quantity_support
>>> with quantity_support():
...     plt.hist(f.distribution, bins=50)
```

(png, svg, pdf)



Using `astropy.uncertainty`

Creating Distributions

The most direct way to create a distribution is to use an array or `Quantity` that carries the samples in the *last* dimension:

```
>>> import numpy as np
>>> from astropy import units as u
>>> from astropy import uncertainty as unc
>>> np.random.seed(123456)  # ensures "random" numbers match examples
below
>>> unc.Distribution(np.random.poisson(12, (1000)))
NdarrayDistribution([..., 12,...]) with n_samples=1000
>>> pq = np.random.poisson([1, 5, 30, 400], (1000, 4)).T * u.ct #
note the transpose, required to get the sampling on the *last* axis
>>> distr = unc.Distribution(pq)
>>> distr
<QuantityDistribution [[...],
           [...],
           [...],
           [...]] ct with n_samples=1000>
```

Note the distinction for these two distributions: the first is built from an array and therefore does not have `Quantity` attributes like `unit`, while the latter does have these attributes. This is reflected in how they interact with other objects, for example, the `NdarrayDistribution` will not combine with **`Quantity`** objects containing units.

For commonly used distributions, helper functions exist to make creating them more convenient. The examples below demonstrate several equivalent ways to create a normal/Gaussian distribution:

```
>>> center = [1, 5, 30, 400]
>>> n_distr = unc.normal(center*u.kpc, std=[0.2, 1.5, 4, 1]*u.kpc,
n_samples=1000)
>>> n_distr = unc.normal(center*u.kpc, var=[0.04, 2.25, 16,
1]*u.kpc**2, n_samples=1000)
>>> n_distr = unc.normal(center*u.kpc, ivar=[25, 0.44444444, 0.625,
1]*u.kpc**-2, n_samples=1000)
>>> n_distr.distribution.shape
(4, 1000)
>>> unc.normal(center*u.kpc, std=[0.2, 1.5, 4, 1]*u.kpc,
n_samples=100).distribution.shape
(4, 100)
>>> unc.normal(center*u.kpc, std=[0.2, 1.5, 4, 1]*u.kpc,
n_samples=20000).distribution.shape
(4, 20000)
```

Additionally, Poisson and uniform **`Distribution`** creation functions exist:

```
>>> unc.poisson(center*u.count, n_samples=1000)
<QuantityDistribution [[...],
             [...],
             [...],
             [...]] ct with n_samples=1000>
>>> uwidth = [10, 20, 10, 55]*u.pc
>>> unc.uniform(center=center*u.kpc, width=uwidth, n_samples=1000)
<QuantityDistribution [[...],
             [...],
             [...],
             [...]] kpc with n_samples=1000>
>>> unc.uniform(lower=center*u.kpc - uwidth/2,  upper=center*u.kpc +
uwidth/2, n_samples=1000)
<QuantityDistribution [[...],
             [...],
             [...],
             [...]] kpc with n_samples=1000>
```

Users are free to create their own distribution classes following similar patterns.

Using Distributions

This object now acts much like a **`Quantity`** or **`numpy.ndarray`** for all but the non-sampled dimension, but with additional statistical operations that work on the sampled distributions:

```
>>> distr.shape
(4,)
>>> distr.size
4
>>> distr.unit
Unit("ct")
>>> distr.n_samples
1000
>>> distr.pdf_mean()
<Quantity [  0.998,   5.017,  30.085, 400.345] ct>
>>> distr.pdf_std()
<Quantity [ 0.97262326,  2.32222114,  5.47629208, 20.6328373 ] ct>
>>> distr.pdf_var()
<Quantity [  0.945996,   5.392711,  29.989775, 425.713975] ct2>
>>> distr.pdf_median()
<Quantity [   1.,   5.,  30., 400.] ct>
>>> distr.pdf_mad()  # Median absolute deviation
<Quantity [ 1.,  2.,  4., 14.] ct>
>>> distr.pdf_smad()  # Median absolute deviation, rescaled to match
std for normal
<Quantity [ 1.48260222,  2.96520444,  5.93040887, 20.75643106] ct>
>>> distr.pdf_percentiles([10, 50, 90])
<Quantity [[  0. ,   2. ,  23. , 374. ],
           [  1. ,   5. ,  30. , 400. ],
           [  2. ,   8. ,  37.1, 427. ]] ct>
>>> distr.pdf_percentiles([.1, .5, .9]*u.dimensionless_unscaled)
<Quantity [[  0. ,   2. ,  23. , 374. ],
          [  1. ,   5. ,  30. , 400. ],
          [  2. ,   8. ,  37.1, 427. ]] ct>
```

If need be, the underlying array can then be accessed from the `distribution` attribute:

```
>>> distr.distribution
<Quantity [[...1...],
           [...5...],
           [...27...],
           [...405...]] ct>
>>> distr.distribution.shape
(4, 1000)
```

A **`Quantity`** distribution interacts naturally with non-**`Distribution`** **`Quantity`** objects, assuming the **`Quantity`** is a Dirac delta distribution:

```
>>> distr_in_kpc = distr * u.kpc/u.count  # for the sake of round numbers in examples
>>> distrplus = distr_in_kpc + [2000,0,0,500]*u.pc
>>> distrplus.pdf_median()
<Quantity [   3. ,    5. ,   30. , 400.5] kpc>
>>> distrplus.pdf_var()
<Quantity [  0.945996,   5.392711,  29.989775, 425.713975] kpc2>
```

It also operates as expected with other distributions (but see below for a discussion of covariances):

```
>>> another_distr = unc.Distribution((np.random.randn(1000,4)*[1000,.01 , 3000, 10] + [2000, 0, 0, 500]).T * u.pc)
>>> combined_distr = distr_in_kpc + another_distr
>>> combined_distr.pdf_median()
<Quantity [  3.01847755,   4.99999576,  29.60559788, 400.49176321] kpc>
>>> combined_distr.pdf_var()
<Quantity [  1.8427705 ,   5.39271147,  39.5343726 , 425.71324244] kpc2>
```

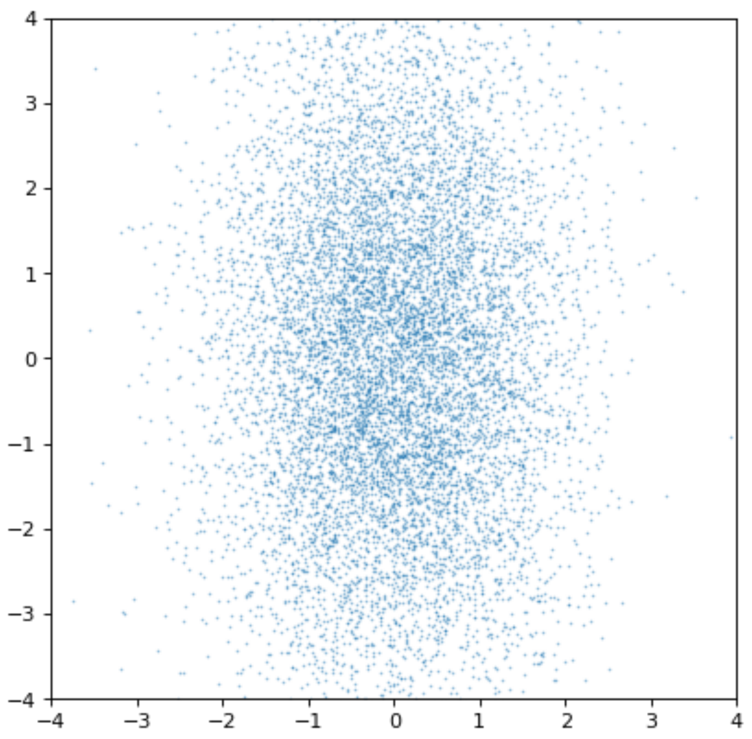
Covariance in Distributions and Discrete Sampling Effects

One of the main applications for distributions is uncertainty propagation, which critically requires proper treatment of covariance. This comes naturally in the Monte Carlo sampling approach used by the **Distribution** class, as long as proper care is taken with sampling error.

To start with a basic example, two un-correlated distributions should produce an un-correlated joint distribution plot:

```
>>> import numpy as np
>>> np.random.seed(12345)  # produce repeatable plots
>>> from astropy import units as u
>>> from astropy import uncertainty as unc
>>> from matplotlib import pyplot as plt
>>> n1 = unc.normal(center=0., std=1, n_samples=10000)
>>> n2 = unc.normal(center=0., std=2, n_samples=10000)
>>> plt.scatter(n1.distribution, n2.distribution, s=2, lw=0, alpha=.5)
>>> plt.xlim(-4, 4)
>>> plt.ylim(-4, 4)
```

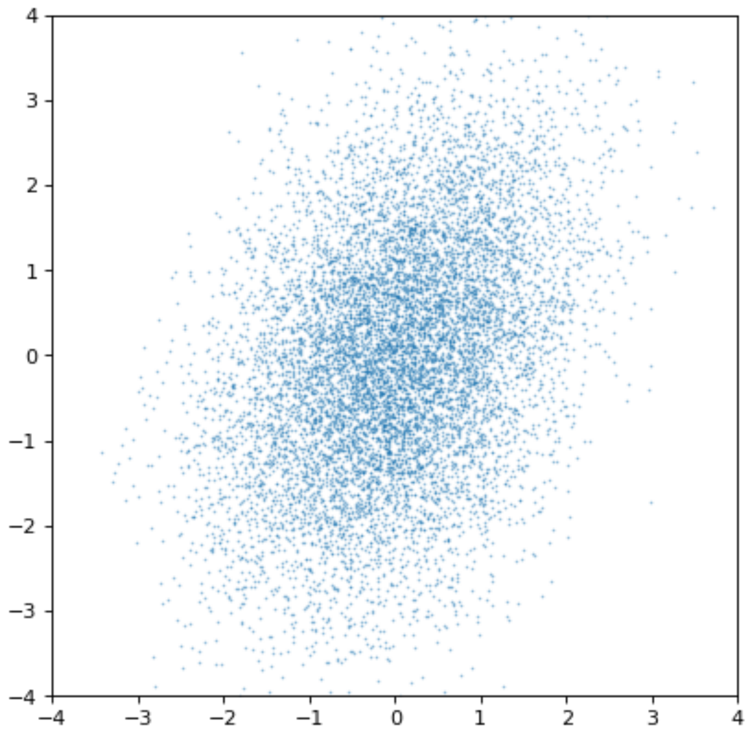
(png, svg, pdf)



Indeed, the distributions are independent. If we instead construct a covariant pair of Gaussians, it is immediately apparent:

```
>>> ncov = np.random.multivariate_normal([0, 0], [[1, .5], [.5, 2]],
size=10000)
>>> n1 = unc.Distribution(ncov[:, 0])
>>> n2 = unc.Distribution(ncov[:, 1])
>>> plt.scatter(n1.distribution, n2.distribution, s=2, lw=0,
alpha=.5)
>>> plt.xlim(-4, 4)
>>> plt.ylim(-4, 4)
```

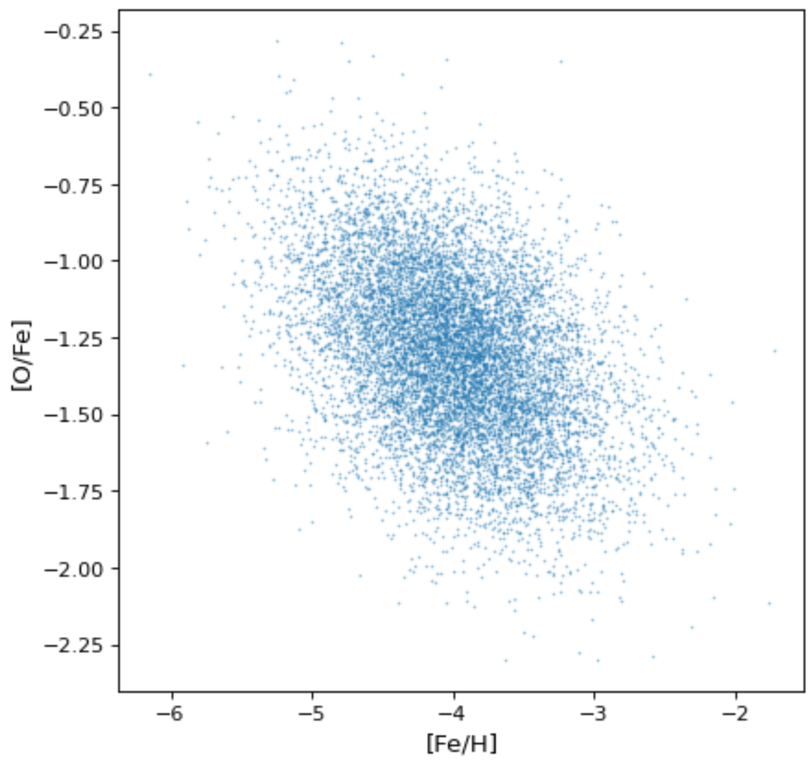
(png, svg, pdf)



Most importantly, the proper correlated structure is preserved or generated as expected by appropriate arithmetic operations. For example, ratios of uncorrelated normal distribution gain covariances if the axes are not independent, as in this simulation of iron, hydrogen, and oxygen abundances in a hypothetical collection of stars:

```
>>> fe_abund = unc.normal(center=-2, std=.25, n_samples=10000)
>>> o_abund = unc.normal(center=-6., std=.5, n_samples=10000)
>>> h_abund = unc.normal(center=-0.7, std=.1, n_samples=10000)
>>> feh = fe_abund - h_abund
>>> ofe = o_abund - fe_abund
>>> plt.scatter(ofe.distribution, feh.distribution, s=2, lw=0,
alpha=.5)
>>> plt.xlabel('[Fe/H]')
>>> plt.ylabel('[O/Fe]')
```

(png, svg, pdf)

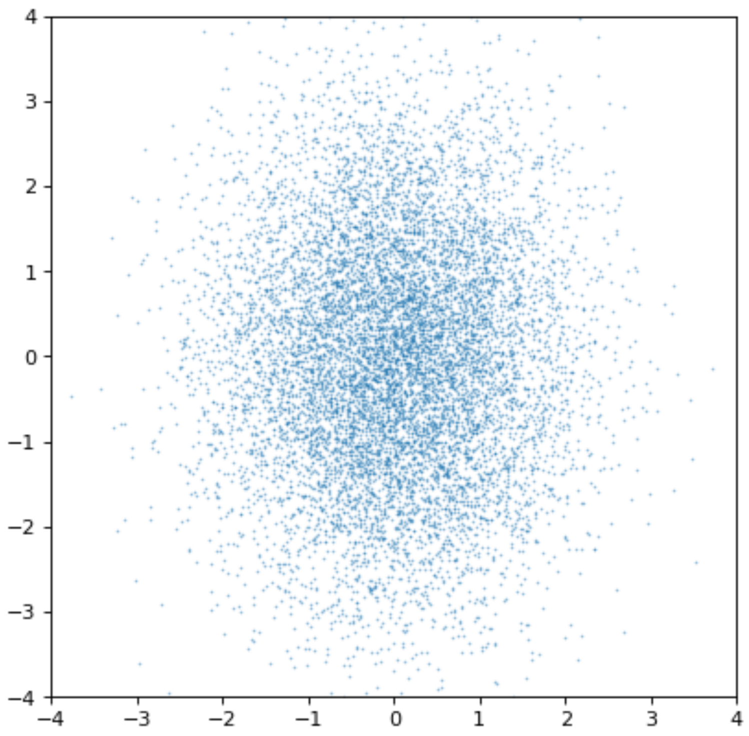


This demonstrates that the correlations naturally arise from the variables, but there is no need to explicitly account for it: the sampling process naturally recovers correlations that are present.

An important note of warning, however, is that the covariance is only preserved if the sampling axes are exactly matched sample by sample. If they are not, all covariance information is (silently) lost:

```
>>> n2_wrong = unc.Distribution(ncov[::-1, 1])  #reverse the sampling axis order
>>> plt.scatter(n1.distribution, n2_wrong.distribution, s=2, lw=0, alpha=.5)
>>> plt.xlim(-4, 4)
>>> plt.ylim(-4, 4)
```

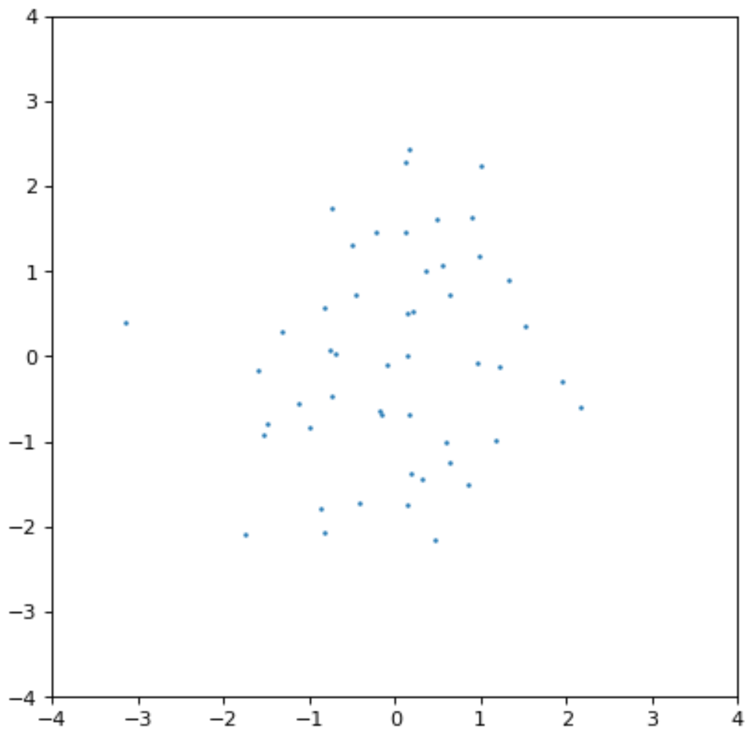
(png, svg, pdf)



Moreover, an insufficiently sampled distribution may give poor estimates or hide correlations. The example below is the same as the covariant Gaussian example above, but with 200x fewer samples:

```
>>> ncov = np.random.multivariate_normal([0, 0], [[1, .5], [.5, 2]],
size=50)
>>> n1 = unc.Distribution(ncov[:, 0])
>>> n2 = unc.Distribution(ncov[:, 1])
>>> plt.scatter(n1.distribution, n2.distribution, s=5, lw=0)
>>> plt.xlim(-4, 4)
>>> plt.ylim(-4, 4)
>>> np.cov(n1.distribution, n2.distribution)
array([[1.04667972, 0.19391617],
       [0.19391617, 1.50899902]])
```

(png, svg, pdf)



The covariance structure is much less apparent by eye, and this is reflected in significant discrepancies between the input and output covariance matrix. In general this is an intrinsic trade-off using sampled distributions: a smaller number of samples is computationally more efficient, but leads to larger uncertainties in any of the relevant quantities. These tend to be of order \(\sqrt{n_{\rm samples}}\) in any derived quantity, but that depends on the complexity of the distribution in question.

Reference/API

astropy.uncertainty Package

This sub-package contains classes and functions for creating distributions that work similar to `Quantity` or array objects, but can propogate uncertainties.

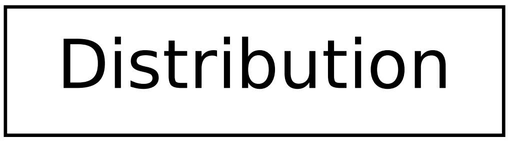
Functions

`normal`(center, *[, std, var, ivar, cls])	Create a Gaussian/normal distribution.
`poisson`(center, n_samples[, cls])	Create a Poisson distribution.
`uniform`(*[, lower, upper, center, width, cls])	Create a Uniform distriution from the lower and upper bounds.

Classes

Distribution(samples)	A scalar value or array values with associated uncertainty distribution.

Class Inheritance Diagram



Files, I/O, and Communication

Unified File Read/Write Interface

`astropy` provides a unified interface for reading and writing data in different formats. For many common cases this will streamline the process of file I/O and reduce the need to master the separate details of all of the I/O packages within `astropy`. For details on the implementation see I/O Registry (astropy.io.registry).

Getting Started with Image I/O

Reading and writing image data in the unified I/O interface is supported though the **CCDData** class using FITS file format:

```
>>> # Read CCD image
>>> ccd = CCDData.read('image.fits')
```

```
>>> # Write back CCD image
>>> ccd.write('new_image.fits')
```

Note that the unit is stored in the `BUNIT` keyword in the header on saving, and is read from the header if it is present.

Detailed help on the available keyword arguments for reading and writing can be obtained via the `help()` method as follows:

```
>>> CCDData.read.help('fits')  # Get help on the CCDData FITS reader
>>> CCDData.writer.help('fits')  # Get help on the CCDData FITS
writer
```

Getting Started with Table I/O

The **`Table`** class includes two methods, **`read()`** and **`write()`**, that make it possible to read from and write to files. A number of formats are automatically supported (see Built-in table readers/writers) and new file formats and extensions can be registered with the **`Table`** class (see I/O Registry (astropy.io.registry)).

Examples

To use this interface, first import the **`Table`** class, then call the **`Table read()`** method with the name of the file and the file format, for instance `'ascii.daophot'`:

```
>>> from astropy.table import Table
>>> t = Table.read('photometry.dat', format='ascii.daophot')
```

It is possible to load tables directly from the Internet using URLs. For example, download tables from Vizier catalogues in CDS format (`'ascii.cds'`):

```
>>> t = Table.read("ftp://cdsarc.u-strasbg.fr/pub/cats/VII/253
/snrs.dat",
...         readme="ftp://cdsarc.u-strasbg.fr/pub/cats/VII/253
/ReadMe",
...         format="ascii.cds")
```

For certain file formats the format can be automatically detected, for example, from the filename extension:

```
>>> t = Table.read('table.tex')
```

For writing a table, the format can be explicitly specified:

```
>>> t.write(filename, format='latex')
```

As for the **`read()`** method, the format may be automatically identified in some cases.

The underlying file handler will also automatically detect various compressed data formats and transparently uncompress them as far as supported by the Python installation (see **`get_readable_fileobj()`**).

For writing, you can also specify details about the Table serialization methods via the `serialize_method` keyword argument. This allows fine control of the way to write out certain columns, for instance writing an ISO format Time column as a pair of JD1/JD2 floating point values (for full resolution) or as a formatted ISO date string.

Getting Help on Readers and Writers

Each file format is handled by a specific reader or writer, and each of those functions will have its own set of arguments. For examples of this see the section Built-in table readers/writers. This section also provides the full list of choices for the `format` argument.

To get help on the available arguments for each format, use the `help()` method of the **`read`** or **`write`** methods. Each of these calls prints a long help document which is divided into two sections, the generic read/write documentation (common to any call) and the format-specific documentation. For ASCII tables, the format-specific documentation includes the generic **`astropy.io.ascii`** package interface and then a description of the particular ASCII sub-format.

In the examples below we do not show the long output:

```
>>> Table.read.help('fits')
>>> Table.read.help('ascii')
>>> Table.read.help('ascii.latex')
>>> Table.write.help('hdf5')
>>> Table.write.help('csv')
```

Command-Line Utility

For convenience, the command-line tool `showtable` can be used to print the content of tables for the formats supported by the unified I/O interface.

Example

To view the contents of a table on the command line:

```
$ showtable astropy/io/fits/tests/data/table.fits

 target V_mag
------- -----
NGC1001  11.1
NGC1002  12.3
NGC1003  15.2
```

To get full documentation on the usage and available options, do `showtable --help`.

Built-In Table Readers/Writers

The **`Table`** class has built-in support for various input and output formats including ASCII Formats, -FITS, HDF5, Pandas, and VO Tables.

A full list of the supported formats and corresponding classes is shown in the table below. The `Write` column indicates those formats that support write functionality, and the `Suffix` column indicates the filename suffix indicating a particular format. If the value of `Suffix` is `auto`, the format is auto-detected from the file itself. Not all formats support auto- detection.

Format	Write	Suffix	Description
ascii	Yes		ASCII table in any supported format (uses guessing)
ascii.aastex	Yes		`AASTex`: AASTeX deluxetable used for AAS journals
ascii.basic	Yes		`Basic`: Basic table with custom delimiters
ascii.cds	No		`Cds`: CDS format table
ascii.commented_header	Yes		`CommentedHeader`: Column names in a commented line
ascii.csv	Yes	.csv	`Csv`: Basic table with comma-separated values
ascii.daophot	No		`Daophot`: IRAF DAOphot format table
ascii.ecsv	Yes	.ecsv	`Ecsv`: Basic table with Enhanced CSV (supporting metadata)
ascii.fixed_width	Yes		`FixedWidth`: Fixed width
ascii.fixed_width_no_header	Yes		`FixedWidthNoHeader`: Fixed width with no header
ascii.fixed_width_two_line	Yes		`FixedWidthTwoLine`: Fixed width with second header line
ascii.html	Yes	.html	`HTML`: HTML table
ascii.ipac	Yes		`Ipac`: IPAC format table
ascii.latex	Yes	.tex	`Latex`: LaTeX table
ascii.no_header	Yes		`NoHeader`: Basic table with no headers
ascii.rdb	Yes	.rdb	`Rdb`: Tab-separated with a type definition header line
ascii.rst	Yes	.rst	`RST`: reStructuredText simple format table
ascii.sextractor	No		`SExtractor`: SExtractor format table
ascii.tab	Yes		`Tab`: Basic table with tab-separated values
fits	Yes	auto	`fits`: Flexible Image Transport System file
hdf5	Yes	auto	HDF5: Hierarchical Data Format binary file

Format	Write	Suffix	Description
pandas.csv	Yes		Wrapper around `pandas.read_csv()` and `pandas.to_csv()`
pandas.fwf	No		Wrapper around `pandas.read_fwf()` (fixed width format)
pandas.html	Yes		Wrapper around `pandas.read_html()` and `pandas.to_html()`
pandas.json	Yes		Wrapper around `pandas.read_json()` and `pandas.to_json()`
votable	Yes	auto	**`votable`**: Table format used by Virtual Observatory (VO) initiative

ASCII Formats

The **`read()`** and **`write()`** methods can be used to read and write formats supported by **`astropy.io.ascii`**.

Use `format='ascii'` in order to interface to the generic **`read()`** and **`write()`** functions from **`astropy.io.ascii`**. When reading a table, this means that all supported ASCII table formats will be tried in order to successfully parse the input.

Examples

To read and write formats supported by **`astropy.io.ascii`**:

```
>>> t = Table.read('astropy/io/ascii/tests/t/latex1.tex',
format='ascii')
>>> print(t)
cola colb colc
---- ---- ----
   a    1    2
   b    3    4
```

When writing a table with `format='ascii'` the output is a basic character-delimited file with a single header line containing the column names.

All additional arguments are passed to the **`astropy.io.ascii`** **`read()`** and **`write()`** functions. Further details are available in the sections on Parameters for read() and Parameters for write(). For example, to change the column delimiter and the output format for the `colc` column use:

```
>>> t.write(sys.stdout, format='ascii', delimiter='|', formats=
{'colc': '%0.2f'})
cola|colb|colc
a|1|2.00
b|3|4.00
```

Note

When specifying an ASCII table format using the unified interface, the format name is prefixed with `ascii` in order to identify the format as ASCII-based. Compare the table above to the **`astropy.io.ascii`** list of supported formats where the prefix is not needed. Therefore the following are equivalent:

```
  >>> dat = ascii.read('file.dat', format='daophot')
  >>> dat = Table.read('file.dat', format='ascii.daophot')

For compatibility with ``astropy`` version 0.2 and earlier, the
following
format values are also allowed in ``Table.read()``: ``daophot``,
``ipac``,
``html``, ``latex``, and ``rdb``.
```

Attention

ECSV is recommended

For writing and reading tables to ASCII in a way that fully reproduces the table data, types, and metadata (i.e., the table will “round-trip”), we highly recommend using the ECSV Format. This writes the actual data in a space-delimited format (the `basic` format) that any ASCII table reader can parse, but also includes metadata encoded in a comment block that allows full reconstruction of the original columns. This includes support for Mixin Columns (such as **`SkyCoord`** or **`Time`**) and Masked Columns.

FITS

Reading and writing tables in FITS format is supported with `format='fits'`. In most cases, existing FITS files should be automatically identified as such based on the header of the file, but if not, or if writing to disk, then the format should be explicitly specified.

Reading

If a FITS table file contains only a single table, then it can be read in with:

```
>>> from astropy.table import Table
>>> t = Table.read('data.fits')
```

If more than one table is present in the file, you can select the HDU as follows:

```
>>> t = Table.read('data.fits', hdu=3)
```

In this case if the `hdu` argument is omitted, then the first table found will be read in and a warning will be emitted:

```
>>> t = Table.read('data.fits')
WARNING: hdu= was not specified but multiple tables are present,
reading in first available table (hdu=1) [astropy.io.fits.connect]
```

You can also read a table from the HDUs of an in-memory FITS file. This will round-trip any Mixin Columns that were written to that HDU, using the header information to reconstruct them:

```
>>> hdulist = astropy.io.fits.open('data.fits')
>>> t = Table.read(hdulist[1])
```

Writing

To write a table `t` to a new file:

```
>>> t.write('new_table.fits')
```

If the file already exists and you want to overwrite it, then set the `overwrite` keyword:

```
>>> t.write('existing_table.fits', overwrite=True)
```

At this time there is no support for appending an HDU to an existing file or writing multi-HDU files using the Table interface. Instead, you can use the convenience function **`table_to_hdu()`** to create a single binary table HDU and insert or append that to an existing **`HDUList`**.

As of `astropy` version 3.0 there is support for writing a table which contains Mixin Columns such as **`Time`** or **`SkyCoord`**. This uses FITS `COMMENT` cards to capture additional information needed order to fully reconstruct the mixin columns when reading back from FITS. The information is a Python **dict** structure which is serialized using YAML.

Keywords

The FITS keywords associated with an HDU table are represented in the `meta` ordered dictionary attribute of a Table. After reading a table you can view the available keywords in a readable format using:

```
>>> for key, value in t.meta.items():
...     print('{0} = {1}'.format(key, value))
```

This does not include the “internal” FITS keywords that are required to specify the FITS table properties (e.g., `NAXIS`, `TTYPE1`). `HISTORY` and `COMMENT` keywords are treated specially and are returned as a list of values.

Conversely, the following shows examples of setting user keyword values for a table `t`:

```
>>> t.meta['MY_KEYWD'] = 'my value'
>>> t.meta['COMMENT'] = ['First comment', 'Second comment', 'etc']
>>> t.write('my_table.fits', overwrite=True)
```

The keyword names (e.g., `MY_KEYWD`) will be automatically capitalized prior to writing.

At this time, the `meta` attribute of the **`Table`** class is an ordered dictionary and does not fully represent the structure of a FITS header (for example, keyword comments are dropped).

TDISPn Keyword

TDISPn FITS keywords will map to and from the **`Column`** `format` attribute if the display format is convertible to and from a Python display format. Below are the rules used for both conversion directions.

TDISPn to Python format string
TDISPn format characters are defined in the table below.

Format	Description

Format	Description
Aw	Character
Lw	Logical
Iw.m	Integer
Bw.m	Binary, integers only
Ow.m	Octal, integers only
Zw.m	Hexadecimal, integers only
Fw.d	Floating-point, fixed decimal notation
Ew.dEe	Floating-point, exponential notation
ENw.d	Engineering; E format with exponent multiple of three
ESw.d	Scientific; same as EN but non-zero leading digit if not zero
Gw.dEe	General; appears as F if significance not lost, also E
Dw.dEe	Floating-point, exponential notation, double precision

Where w is the width in characters of displayed values, m is the minimum number of digits displayed, d is the number of digits to the right of decimal, and e is the number of digits in the exponent. The .m and Ee fields are optional.

The A (character), L (logical), F (floating point), and G (general) display formats can be directly translated to Python format strings. The other formats need to be modified to match Python display formats.

For the integer formats (I, B, O, and Z), the width (w) value is used to add space padding to the left of the column value. The minimum number (m) value is not used. For the E, G, D, EN, and ES formats (floating point exponential) the width (w) and precision (d) are both used, but the exponential (e) is not used.

Python format string to TDISPn

The conversion from Python format strings back to TDISPn is slightly more complicated.

Python strings map to the TDISP format A if the Python formatting string does not contain right space padding. It will accept left space padding. The same applies to the logical format L.

The integer formats (decimal integer, binary, octal, hexidecimal) map to the I, B, O, and Z TDISP formats respectively. Integer formats do not accept a zero padded format string or a format string with no left padding defined (a width is required in the TDISP format standard for the Integer formats).

For all float and exponential values, zero padding is not accepted. There must be at least a width or precision defined. If only a width is defined, there is no precision set for the TDISPn format. If only a precision is defined, the width is set to the precision plus an extra padding value depending on format type, and both are set in the TDISPn format. Otherwise, if both a width and precision are present they are both set in the TDISPn format. A Python `f` or `F` map to TDISP F format. The Python `g` or `G` map to TDISP G format. The Python `e` and `E` map to TDISP E format.

Masked Columns

Tables that contain **`MaskedColumn`** columns can be written to FITS. By default this will replace the masked data elements with certain sentinel values according to the FITS standard:

- `NaN` for float columns.
- Value of `TNULLn` for integer columns, as defined by the column `fill_value` attribute.
- Null string for string columns (not currently implemented).

When the file is read back those elements are marked as masked in the returned table, but see issue #4708 for problems in all three cases.

The FITS standard has a few limitations:

- Not all data types are supported (e.g., logical / boolean).
- Integer columns require picking one value as the NULL indicator. If all possible values are represented in valid data (e.g., an unsigned int columns with all 256 possible values in valid data), then there is no way to represent missing data.
- The masked data values are permanently lost, precluding the possibility of later unmasking the values.

`astropy` provides a work-around for this limitation that users can choose to use. The key part is to use the `serialize_method='data_mask'` keyword argument when writing the table. This tells the FITS writer to split each masked column into two separate columns, one for the data and one for the mask. When it gets read back that process is reversed and the two columns are merged back into one masked column.

```
>>> from astropy.table.table_helpers import simple_table
>>> t = simple_table(masked=True)
>>> t['d'] = [False, False, True]
>>> t['d'].mask = [True, False, False]
>>> t
<Table masked=True length=3>
  a      b      c      d
```

```
int64 float64 str1  bool
----- ------- ---- -----
   --     1.0    c    --
    2     2.0   -- False
    3      --    e  True
```

```
>>> t.write('data.fits', serialize_method='data_mask',
overwrite=True)
>>> Table.read('data.fits')
<Table masked=True length=3>
  a      b      c      d
int64 float64 bytes1  bool
----- ------- ------ -----
   --     1.0      c    --
    2     2.0     -- False
    3      --      e  True
```

Warning

This option goes outside of the established FITS standard for representing missing data, so users should be careful about choosing this option, especially if other (non-`astropy`) users will be reading the file(s). Behind the scenes, `astropy` is converting the masked columns into two distinct data and mask columns, then writing metadata into `COMMENT` cards to allow reconstruction of the original data.

`astropy` Native Objects (Mixin Columns)

It is possible to store not only standard **`Column`** objects to a FITS table HDU, but also any `astropy` native objects (Mixin Columns) within a **`Table`** or **`QTable`**. This includes **`Time`**, **`Quantity`**, **`SkyCoord`**, and many others.

In general, a mixin column may contain multiple data components as well as object attributes beyond the standard Column attributes like `format` or `description`. Abiding by the rules set by the FITS standard requires the mapping of these data components and object attributes to the appropriate FITS table columns and keywords. Thus, a well defined protocol has been developed to allow the storage of these mixin columns in FITS while allowing the object to “round-trip” through the file with no loss of data or attributes.

Quantity

A **`Quantity`** mixin column in a **`QTable`** is represented in a FITS table using the `TUNITn` FITS column keyword to incorporate the unit attribute of Quantity.

For example:

```
>>> from astropy.table import QTable
>>> import astropy.units as u
>>> t = QTable([[1, 2] * u.angstrom)])
>>> t.write('my_table.fits', overwrite=True)
>>> qt = QTable.read('my_table.fits')
>>> qt
<QTable length=2>
  col0
Angstrom
float64
--------
     1.0
     2.0
```

Time

`astropy` provides the following features for reading and writing `Time`:

- Writing and reading **Time** Table columns to and from FITS tables.
- Reading time coordinate columns in FITS tables (compliant with the time standard) as **Time** Table columns.

Writing and reading `astropy` Time columns

By default, a **Time** mixin column within a **Table** or **QTable** will be written to FITS in full precision. This will be done using the FITS time standard by setting the necessary FITS header keywords.

The default behavior for reading a FITS table into a **Table** has historically been to convert all FITS columns to **Column** objects, which have closely matching properties. For some columns, however, closer native `astropy` representations are possible, and you can indicate these should be used by passing `astropy_native=True` (for backwards compatibility, this is not done by default). This will convert columns conforming to the FITS time standard to **Time** instances, avoiding any loss of precision.

Example

To read a FITS table into **Table**:

```
>>> from astropy.time import Time
>>> from astropy.table import Table
>>> from astropy.coordinates import EarthLocation
>>> t = Table()
>>> t['a'] = Time([100.0, 200.0], scale='tt', format='mjd',
...               location=EarthLocation(-2446354, 4237210, 4077985,
unit='m'))
```

```
>>> t.write('my_table.fits', overwrite=True)
>>> tm = Table.read('my_table.fits', astropy_native=True)
>>> tm['a']
<Time object: scale='tt' format='jd' value=[ 2400100.5  2400200.5]>
>>> tm['a'].location
<EarthLocation (-2446354.,  4237210.,  4077985.) m>
>>> all(tm['a'] == t['a'])
True
```

The same will work with `QTable`.

In addition to binary table columns, various global time informational FITS keywords are treated specially with `astropy_native=True`. In particular, the keywords `DATE`, `DATE-*` (ISO 8601 datetime strings), and the `MJD-*` (MJD date values) will be returned as `Time` objects in the Table `meta`. For more details regarding the FITS time paper and the implementation, refer to FITS Tables with Time Columns.

Since not all FITS readers are able to use the FITS time standard, it is also possible to store **`Time`** instances using the **`_time_format`**. For this case, none of the special header keywords associated with the FITS time standard will be set. When reading this back into `astropy`, the column will be an ordinary Column instead of a **`Time`** object. See the Details section below for an example.

Reading FITS standard compliant time coordinate columns in binary tables

Reading FITS files which are compliant with the FITS time standard is supported by `astropy` by following the multifarious rules and conventions set by the standard. The standard was devised in order to describe time coordinates in an unambiguous and comprehensive manner and also to provide flexibility for its multiple use cases. Thus, while reading time coordinate columns in FITS- compliant files, multiple aspects of the standard are taken into consideration.

Time coordinate columns strictly compliant with the two-vector JD subset of the standard (described in the Details section below) can be read as native **`Time`** objects. The other subsets of the standard are also supported by `astropy`; a thorough examination of the FITS standard time- related keywords is done and the time data is interpreted accordingly.

The standard describes the various components in the specification of time:

- Time coordinate frame
- Time unit
- Corrections, errors, etc.
- Durations

The keywords used to specify times define these components. Using these keywords, time coordinate columns are identified and read as **Time** objects. Refer to FITS Tables with Time Columns for the specification of these keywords and their description.

There are two aspects of the standard that require special attention due to the subtleties involved while handling them. These are:

- Column named TIME with time unit

A common convention found in existing FITS files is that a FITS binary table column with `TTYPEn = ‘TIME’` represents a time coordinate column. Many astronomical data files, including official data products from major observatories, follow this convention that predates the FITS standard. The FITS time standard states that such a column will be controlled by the global time reference frame keywords, and this will still be compliant with the present standard.

Using this convention which has been incorporated into the standard, `astropy` can read time coordinate columns from all such FITS tables as native **Time** objects. Common examples of FITS files following this convention are Chandra, XMM, and HST files.

Examples

The following is an example of a Header extract of a Chandra event list:

```
COMMENT      ---------- Globally valid key words ----------------
DATE     = '2016-01-27T12:34:24' / Date and time of file creation
TIMESYS = 'TT      '           / Time system
MJDREF   =  5.0814000000000E+04 / [d] MJD zero point for times
TIMEUNIT= 's       '           / Time unit
TIMEREF = 'LOCAL   '           / Time reference (barycenter/local)

COMMENT      ---------- Time Column -----------------------
TTYPE1   = 'time    '          / S/C TT corresponding to mid-exposure
TFORM1   = '1D      '          / format of field
TUNIT1   = 's       '
```

When reading such a FITS table with `astropy_native=True`, `astropy` checks whether the name of a column is “TIME”/ “time” (`TTYPEn = ‘TIME’`) and whether its unit is a FITS recognized time unit (`TUNITn` is a time unit).

For example, reading a Chandra event list which has the above mentioned header and the time coordinate column `time` as `[1, 2]` will give:

```
>>> from astropy.table import Table
>>> from astropy.time import Time, TimeDelta
>>> from astropy.utils.data import get_pkg_data_filename
>>> chandra_events = get_pkg_data_filename('data/chandra_time.fits',
...
package='astropy.io.fits.tests')
>>> native = Table.read(chandra_events, astropy_native=True)
>>> native['time']
<Time object: scale='tt' format='mjd' value=[57413.76033393
57413.76033393]>
>>> non_native = Table.read(chandra_events)
>>> # MJDREF  =  5.0814000000000E+04, TIMESYS = 'TT'
>>> ref_time = Time(non_native.meta['MJDREF'], format='mjd',
...                     scale=non_native.meta['TIMESYS'].lower())
>>> # TTYPE1  = 'time', TUNIT1 = 's'
>>> delta_time = TimeDelta(non_native['time'])
>>> all(ref_time + delta_time == native['time'])
True
```

By default, FITS table columns will be read as standard `Column` objects without taking the FITS time standard into consideration.

- String time column in ISO 8601 Datetime format

FITS uses a subset of ISO 8601 (which in itself does not imply a particular timescale) for several time-related keywords, such as DATE-xxx. Following the FITS standard, its values must be written as a character string in the following `datetime` format:

```
[+/-C]CCYY-MM-DD[Thh:mm:ss[.s...]]
```

A time coordinate column can be constructed using this representation of time. The following is an example of an ISO 8601 `datetime` format time column:

```
TIME
----
1999-01-01T00:00:00
1999-01-01T00:00:40
1999-01-01T00:01:06
.
.
.
1999-01-20T01:10:00
```

The criteria for identifying a time coordinate column in ISO 8601 format is as follows:

A time column is identified using the time coordinate frame keywords as described in FITS Tables with Time Columns. Once it has been identified, its datatype is checked in order to determine its representation format. Since ISO 8601 `datetime` format is the only string representation of time, a time coordinate column having string datatype will be automatically read as a **`Time`** object with `format='fits'` (‘fits’ represents the FITS ISO 8601 format).

As this format does not imply a particular timescale, it is determined using the timescale keywords in the header (`TCTYP` or `TIMESYS`) or their defaults. The other time coordinate information is also determined in the same way, using the time coordinate frame keywords. All ISO 8601 times are relative to a globally accepted zero point (year 0 corresponds to 1 BCE) and are thus not relative to the reference time keywords (MJDREF, JDREF, or DATEREF). Hence, these keywords will be ignored while dealing with ISO 8601 time columns.

Note

Reading FITS files with time coordinate columns *may* fail. `astropy` supports a large subset of these files, but there are still some FITS files which are not compliant with any aspect of the standard. If you have such a file, please do not hesitate to let us know (by opening an issue in the issue tracker).

Also, reading a column having `TTYPEn = ‘TIME’` as **`Time`** will fail if `TUNITn` for the column is not a FITS-recognized time unit.

Details

Time as a dimension in astronomical data presents challenges in its representation in FITS files. The standard has therefore been extended to describe rigorously the time coordinate in the `World Coordinate System` framework. Refer to FITS WCS paper IV for details.

Allowing `Time` columns to be written as time coordinate columns in FITS tables thus involves storing time values in a way that ensures retention of precision and mapping the associated metadata to the relevant FITS keywords.

In accordance with the standard, which states that in binary tables one may use pairs of doubles, the `astropy` Time column is written in such a table as a vector of two doubles `(TFORMn = ‘2D’) (jd1, jd2)` where `JD = jd1 + jd2`. This reproduces the time values to double-double precision and is the “lossless” version, exploiting the higher precision provided in binary tables. Note that `jd1` is always a half-integer or integer, while `abs(jd2) < 1`. “Round-tripping” of `astropy`-written FITS binary tables containing time coordinate columns has been partially achieved by mapping selected metadata, `scale` and singular `location` of **`Time`**, to corresponding keywords. Note that the arbitrary metadata allowed in **`Table`** objects within the `meta` dict is not written and will be lost.

Examples

Consider the following Time column:

```
>>> t['a'] = Time([100.0, 200.0], scale='tt', format='mjd')
```

The FITS standard requires an additional translation layer back into the desired format. The Time column `t['a']` will undergo the translation `Astropy Time --> FITS --> Astropy Time` which corresponds to the format conversion `mjd --> (jd1, jd2) --> jd`. Thus, the final conversion from `(jd1, jd2)` will require a software implementation which is fully compliant with the FITS time standard.

Taking this into consideration, the functionality to read/write Time from/to FITS can be explicitly turned off, by opting to store the time representation values in the format specified by the `format` attribute of the **`Time`** column, instead of the `(jd1, jd2)` format, with no extra metadata in the header. This is the "lossy" version, but can help with portability. For the above example, the FITS column corresponding to `t['a']` will then store `[100.0 200.0]` instead of `[[ 2400100.5, 0. ], [ 2400200.5, 0. ]]`. This is done by setting the Table serialization methods for Time columns when writing, as in the following example:

```
>>> from astropy.time import Time
>>> from astropy.table import Table
>>> from astropy.coordinates import EarthLocation
>>> t = Table()
>>> t['a'] = Time([100.0, 200.0], scale='tt', format='mjd')
>>> t.write('my_table.fits', overwrite=True,
...         serialize_method={Time: 'formatted_value'})
>>> tm = Table.read('my_table.fits')
>>> tm['a']
<Column name='a' dtype='float64' length=2>
100.0
200.0
>>> all(tm['a'] == t['a'].value)
True
```

By default, `serialize_method` for Time columns is equal to `'jd1_jd2'`, that is, Time columns will be written in full precision.

Note

The `astropy` **`Time`** object does not precisely map to the FITS time standard.

- FORMAT

The FITS format considers only three formats: ISO 8601, JD, and MJD. `astropy` Time allows for many other formats like `unix` or `cxcsec` for representing the values.

Hence, the `format` attribute of Time is not stored. After reading from FITS the user must set the `format` as desired.

- LOCATION

  In the FITS standard, the reference position for a time coordinate is a scalar expressed via keywords. However, vectorized reference position or location can be supported by the Green Bank Keyword Convention which is a Registered FITS Convention. In `astropy` Time, location can be an array which is broadcastable to the Time values.

  Hence, vectorized `location` attribute of Time is stored and read following this convention.

HDF5

Reading/writing from/to HDF5 files is supported with `format='hdf5'` (this requires h5py to be installed). However, the `.hdf5` file extension is automatically recognized when writing files, and HDF5 files are automatically identified (even with a different extension) when reading in (using the first few bytes of the file to identify the format), so in most cases you will not need to explicitly specify `format='hdf5'`.

Since HDF5 files can contain multiple tables, the full path to the table should be specified via the `path=` argument when reading and writing.

Examples

To read a table called `data` from an HDF5 file named `observations.hdf5`, you can do:

```
>>> t = Table.read('observations.hdf5', path='data')
```

To read a table nested in a group in the HDF5 file, you can do:

```
>>> t = Table.read('observations.hdf5', path='group/data')
```

To write a table to a new file, the path should also be specified:

```
>>> t.write('new_file.hdf5', path='updated_data')
```

It is also possible to write a table to an existing file using `append=True`:

```
>>> t.write('observations.hdf5', path='updated_data', append=True)
```

As with other formats, the `overwrite=True` argument is supported for overwriting existing files. To overwrite only a single table within an HDF5 file that has multiple datasets, use *both* the `overwrite=True` and `append=True` arguments.

Finally, when writing to HDF5 files, the `compression=` argument can be used to ensure that the data is compressed on disk:

```
>>> t.write('new_file.hdf5', path='updated_data', compression=True)
```

Metadata and Mixin Columns

`astropy` tables can contain metadata, both in the table `meta` attribute (which is an ordered dictionary of arbitrary key/value pairs), and within the columns, which each have attributes `unit`, `format`, `description`, and `meta`.

By default, when writing a table to HDF5 the code will attempt to store each key/value pair within the table `meta` as HDF5 attributes of the table dataset. This will fail if the values within `meta` are not objects that can be stored as HDF5 attributes. In addition, if the table columns being stored have defined values for any of the above-listed column attributes, these metadata will *not* be stored and a warning will be issued.

serialize_meta
To enable storing all table and column metadata to the HDF5 file, call the `write()` method with `serialize_meta=True`. This will store metadata in a separate HDF5 dataset, contained in the same file, which is named `<path>.__table_column_meta__`. Here `path` is the argument provided in the call to `write()`:

```
>>> t.write('observations.hdf5', path='data', serialize_meta=True)
```

The table metadata are stored as a dataset of strings by serializing the metadata in YAML following the ECSV header format definition. Since there are

YAML parsers for most common languages, one can easily access and use the table metadata if reading the HDF5 in a non-astropy application.

As of `astropy` 3.0, by specifying `serialize_meta=True` one can also store to HDF5 tables that contain Mixin Columns such as **`Time`** or **`SkyCoord`** columns.

Pandas

`astropy` **`Table`** supports the ability to read or write tables using some of the I/O methods available within pandas. This interface thus provides convenient wrappers to the following functions / methods:

Format name	Data Description	Reader	Writer
`pandas.csv`	CSV	read_csv()	to_csv()
`pandas.json`	JSON	read_json()	to_json()
`pandas.html`	HTML	read_html()	to_html()
`pandas.fwf`	Fixed Width	read_fwf()	

Notes:

- There is no fixed-width writer in pandas.
- Reading HTML requires BeautifulSoup4 and html5lib to be installed.

When reading or writing a table, any keyword arguments apart from the `format` and file name are passed through to pandas, for instance:

```
>>> t.write('data.csv', format='pandas.csv', sep=' ', header=False)
>>> t2 = Table.read('data.csv', format='pandas.csv', sep=' ',
names=['a', 'b', 'c'])
```

JSViewer

Provides an interactive HTML export of a Table, like the **`HTML`** writer but using the DataTables library, which allow to visualize interactively an HTML table (with columns sorting, search, and pagination).

Example

To write a table `t` to a new file:

```
>>> t.write('new_table.html', format='jsviewer')
```

Several additional parameters can be used:

- *table_id*: the HTML ID of the `<table>` tag, defaults to `'table{id}'` where `id` is the ID of the Table object.
- *max_lines*: maximum number of lines.
- *table_class*: HTML classes added to the `<table>` tag, can be useful to customize the style of the table.
- *jskwargs*: additional arguments passed to **`JSViewer`**.
- *css*: CSS style, default to `astropy.table.jsviewer.DEFAULT_CSS`.
- *htmldict*: additional arguments passed to **`HTML`**.

VO Tables

Reading/writing from/to VO table files is supported with `format='votable'`. In most cases, existing VO tables should be automatically identified as such based on the header of the file, but if not, or if writing to disk, then the format should be explicitly specified.

Examples

If a VO table file contains only a single table, then it can be read in with:

```
>>> t = Table.read('aj285677t3_votable.xml')
```

If more than one table is present in the file, an error will be raised, unless the table ID is specified via the `table_id=` argument:

```
>>> t = Table.read('catalog.xml')
Traceback (most recent call last):
...
ValueError: Multiple tables found: table id should be set via the
table_id= argument. The available tables are twomass, spitzer

>>> t = Table.read('catalog.xml', table_id='twomass')
```

To write to a new file, the ID of the table should also be specified (unless `t.meta['ID']` is defined):

```
>>> t.write('new_catalog.xml', table_id='updated_table',
format='votable')
```

When writing, the `compression=True` argument can be used to force compression of the data on disk, and the `overwrite=True` argument can be used to overwrite an existing file.

Table Serialization Methods

`astropy` supports fine-grained control of the way to write out (serialize) the columns in a Table. For instance, if you are writing an ISO format Time column to an ECSV ASCII table file, you may want to write this as a pair of JD1/JD2 floating point values for full resolution (perfect “round-trip”), or as a formatted ISO date string so that the values are easily readable by your other applications.

The default method for serialization depends on the format (FITS, ECSV, HDF5). For instance HDF5 is a binary format and so it would make sense to store a Time object as JD1/JD2, while ECSV is a flat ASCII format and commonly you would want to see the date in the same format as the Time object. The defaults also reflect an attempt to minimize compatibility issues between `astropy` versions. For instance, it was possible to write Time columns to ECSV as formatted strings in a version prior to the ability to write as JD1/JD2 pairs, so the current default for ECSV is to write as formatted strings.

The two classes which have configurable serialization methods are **`Time`** and **`MaskedColumn`**. See the sections on Time Details and Masked columns, respectively, for additional information. The defaults for each format are listed below:

Format	Time	MaskedColumn
FITS	`jd1_jd2`	`null_value`
ECSV	`formatted_value`	`null_value`
HDF5	`jd1_jd2`	`data_mask`
YAML	`jd2_jd2`	—

Examples

Start by making a table with a Time column and masked column:

```
>>> import sys
>>> from astropy.time import Time
>>> from astropy.table import Table, MaskedColumn
```

```
>>> t = Table(masked=True)
>>> t['tm'] = Time(['2000-01-01', '2000-01-02'])
>>> t['mc1'] = MaskedColumn([1.0, 2.0], mask=[True, False])
>>> t['mc2'] = MaskedColumn([3.0, 4.0], mask=[False, True])
>>> t
```

```
<Table masked=True length=2>
           tm              mc1     mc2
         object          float64 float64
----------------------- ------- -------
2000-01-01 00:00:00.000      --     3.0
2000-01-02 00:00:00.000     2.0      --
```

Now specify that you want all **Time** columns written as JD1/JD2 and the `mc1` column written as a data/mask pair and write to ECSV:

```
>>> serialize_method = {Time: 'jd1_jd2', 'mc1': 'data_mask'}
>>> t.write(sys.stdout, format='ascii.ecsv',
serialize_method=serialize_method)
# %ECSV 0.9
 ...
# schema: astropy-2.0
 tm.jd1    tm.jd2  mc1  mc1.mask  mc2
2451544.0    0.5   1.0   True     3.0
2451546.0   -0.5   2.0   False     ""
```

(Spaces added for clarity)

Notice that the `tm` column has been replaced by the `tm.jd1` and `tm.jd2` columns, and likewise a new column `mc1.mask` has appeared and it explicitly contains the mask values. When this table is read back with the `ascii.ecsv` reader then the original columns are reconstructed.

The `serialize_method` argument can be set in two different ways:

- As a single string like `data_mask`. This value then applies to every column, and is a convenient strategy for a masked table with no Time columns.
- As a **dict**, where the key can be either a single column name or a class (as shown in the example above), and the value is the corresponding serialization method.

FITS File Handling (**astropy.io.fits**)

Introduction

The **astropy.io.fits** package provides access to FITS files. FITS (Flexible Image Transport System) is a portable file standard widely used in the astronomy community to store images and tables.

Getting Started

This section provides a quick introduction of using **astropy.io.fits**. The goal is to demonstrate the package's basic features without getting into too much detail. If you are a first time user or have never used `astropy` or PyFITS, this is where you should start. See also the FAQ for answers to common questions and issues.

Note

If you want to read or write a single table in FITS format, the recommended method is via the high-level Unified File Read/Write Interface. In particular see the Unified I/O FITS section.

Reading and Updating Existing FITS Files

Opening a FITS File

Note

The `astropy.io.fits.util.get_testdata_filepath()` function, used in the examples here, is for accessing data shipped with `astropy`. To work with your own data instead, please use **`astropy.io.fits.open()`**, which takes either the relative or absolute path.

Once the **`astropy.io.fits`** package is loaded using the standard convention [1], we can open an existing FITS file:

```
>>> from astropy.io import fits
>>> fits_image_filename =
fits.util.get_testdata_filepath('test0.fits')

>>> hdul = fits.open(fits_image_filename)
```

The **`open()`** function has several optional arguments which will be discussed in a later chapter. The default mode, as in the above example, is “readonly”. The open function returns an object called an **`HDUList`** which is a **`list`**-like collection of HDU objects. An HDU (Header Data Unit) is the highest level component of the FITS file structure, consisting of a header and (typically) a data array or table.

After the above open call, `hdul[0]` is the primary HDU, `hdul[1]` is the first extension HDU, etc. (if there are any extensions), and so on. It should be noted that `astropy` uses zero-based indexing when referring to HDUs and header cards, though the FITS standard (which was designed with Fortran in mind) uses one-based indexing.

The HDUList has a useful method HDUList.info(), which summarizes the content of the opened FITS file:

```
>>> hdul.info()
Filename: ...test0.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU     138   ()
  1  SCI           1 ImageHDU        61   (40, 40)   int16
  2  SCI           2 ImageHDU        61   (40, 40)   int16
  3  SCI           3 ImageHDU        61   (40, 40)   int16
  4  SCI           4 ImageHDU        61   (40, 40)   int16
```

After you are done with the opened file, close it with the HDUList.close() method:

```
>>> hdul.close()
```

You can avoid closing the file manually by using open() as context manager:

```
>>> with fits.open(fits_image_filename) as hdul:
...     hdul.info()
Filename: ...test0.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU     138   ()
  1  SCI           1 ImageHDU        61   (40, 40)   int16
  2  SCI           2 ImageHDU        61   (40, 40)   int16
  3  SCI           3 ImageHDU        61   (40, 40)   int16
  4  SCI           4 ImageHDU        61   (40, 40)   int16
```

After exiting the `with` scope the file will be closed automatically. That is (generally) the preferred way to open a file in Python, because it will close the file even if an exception happens.

If the file is opened with `lazy_load_hdus=False`, all of the headers will still be accessible after the HDUList is closed. The headers and data may or may not be accessible depending on whether the data are touched and if they are memory-mapped; see later chapters for detail.

Working with large files

The open() function supports a `memmap=True` argument that allows the array data of each HDU to be accessed with mmap, rather than being read into memory all at once. This is particularly useful for working with very large arrays that cannot fit entirely into physical memory. Here `memmap=True` by default, and this value is obtained from the configuration item `astropy.io.fits.Conf.use_memmap`.

This has minimal impact on smaller files as well, though some operations, such as reading the array data sequentially, may incur some additional overhead. On 32-bit systems, arrays larger than 2 to 3 GB cannot be mmap'd (which is fine, because by that point you are likely to run out of physical memory anyways), but 64-bit systems are much less limited in this respect.

Warning

When opening a file with `memmap=True`, because of how mmap works this means that when the HDU data is accessed (i.e., `hdul[0].data`) another handle to the FITS file is opened by mmap. This means that even after calling `hdul.close()` the mmap still holds an open handle to the data so that it can still be accessed by unwary programs that were built with the assumption that the .data attribute has all of the data in-memory.

In order to force the mmap to close, either wait for the containing `HDUList` object to go out of scope, or manually call `del hdul[0].data`. (This works so long as there are no other references held to the data array.)

Unsigned integers

Due to the FITS format's Fortran origins, FITS does not natively support unsigned integer data in images or tables. However, there is a common convention to store unsigned integers as signed integers, along with a *shift* instruction (a `BZERO` keyword with value `2 ** (BITPIX - 1)`) to shift up all signed integers to unsigned integers. For example, when writing the value `0` as an unsigned 32-bit integer, it is stored in the FITS file as `-32768`, along with the header keyword `BZERO = 32768`.

`astropy` recognizes and applies this convention by default, so that all data that looks like it should be interpreted as unsigned integers is automatically converted (this applies to both images and tables). In `astropy` versions prior to v1.1.0 this was *not* applied automatically, and it is necessary to pass the argument `uint=True` to **`open()`**. In v1.1.0 or later this is the default.

Even with `uint=False`, the `BZERO` shift is still applied, but the returned array is of "float64" type. To disable scaling/shifting entirely, use `do_not_scale_image_data=True` (see Why is an image containing integer data being converted unexpectedly to floats? in the FAQ for more details).

Working with compressed files

Note

Files that use compressed HDUs within the FITS file are discussed in Compressed Image Data.

The `open()` function will seamlessly open FITS files that have been compressed with gzip, bzip2 or pkzip. Note that in this context we are talking about a FITS file that has been compressed with one of these utilities (e.g., a .fits.gz file).

There are some limitations when working with compressed files. For example, with Zip files that contain multiple compressed files, only the first file will be accessible. Also bzip2 does not support the append or update access modes.

When writing a file (e.g., with the `writeto()` function), compression will be determined based on the filename extension given, or the compression used in a pre-existing file that is being written to.

Working with non-standard files

When `astropy.io.fits` reads a FITS file which does not conform to the FITS standard it will try to make an educated interpretation of non-compliant fields. This may not always succeed and may trigger warnings when accessing headers or exceptions when writing to file. Verification of fields written to an output file can be controlled with the `output_verify` parameter of `open()`. Files opened for reading can be verified and fixed with method `HDUList.verify`. This method is typically invoked after opening the file but before accessing any headers or data:

```
>>> with fits.open(fits_image_filename) as hdul:
...     hdul.verify('fix')
...     data = hdul[1].data
```

In the above example, the call to `hdul.verify("fix")` requests that `astropy.io.fits` fix non-compliant fields and print informative messages. Other options in addition to `"fix"` are described under FITS Verification

See also

FITS Verification.

Working with FITS Headers

As mentioned earlier, each element of an `HDUList` is an HDU object with `.header` and `.data` attributes, which can be used to access the header and data portions of the HDU.

For those unfamiliar with FITS headers, they consist of a list of 80 byte “cards”, where a card contains a keyword, a value, and a comment. The keyword and comment must both be strings, whereas the value can be a string or an integer, floating point number, complex number, or `True` / `False`. Keywords are usually unique within a header, except in a few special cases.

The header attribute is a Header instance, another `astropy` object. To get the value associated with a header keyword, do (à la Python dicts):

```
>>> hdul = fits.open(fits_image_filename)
>>> hdul[0].header['DATE']
'01/04/99'
```

to get the value of the keyword “DATE”, which is a string ‘01/04/99’.

Although keyword names are always in upper case inside the FITS file, specifying a keyword name with `astropy` is case-insensitive for the user’s convenience. If the specified keyword name does not exist, it will raise a **`KeyError`** exception.

We can also get the keyword value by indexing (à la Python lists):

```
>>> hdul[0].header[7]
32768.0
```

This example returns the eighth (like Python lists, it is 0-indexed) keyword’s value — a float — 32768.0.

Similarly, it is possible to update a keyword’s value in `astropy`, either through keyword name or index:

```
>>> hdr = hdul[0].header
>>> hdr['targname'] = 'NGC121-a'
>>> hdr[27] = 99
```

Please note however that almost all application code should update header values via their keyword name and not via their positional index. This is because most FITS keywords may appear at any position in the header.

It is also possible to update both the value and comment associated with a keyword by assigning them as a tuple:

```
>>> hdr = hdul[0].header
>>> hdr['targname'] = ('NGC121-a', 'the observation target')
>>> hdr['targname']
'NGC121-a'
>>> hdr.comments['targname']
'the observation target'
```

Like a dict, you may also use the above syntax to add a new keyword/value pair (and optionally a comment as well). In this case the new card is appended to the end of the header (unless it is a commentary keyword such as COMMENT or HISTORY, in which case it is appended after the last card with that keyword).

Another way to either update an existing card or append a new one is to use the **`Header.set()`** method:

```
>>> hdr.set('observer', 'Edwin Hubble')
```

Comment or history records are added like normal cards, though in their case a new card is always created, rather than updating an existing HISTORY or COMMENT card:

```
>>> hdr['history'] = 'I updated this file 2/26/09'
>>> hdr['comment'] = 'Edwin Hubble really knew his stuff'
>>> hdr['comment'] = 'I like using HST observations'
>>> hdr['history']
I updated this file 2/26/09
>>> hdr['comment']
Edwin Hubble really knew his stuff
I like using HST observations
```

Note: Be careful not to confuse COMMENT cards with the comment value for normal cards.

To update existing COMMENT or HISTORY cards, reference them by index:

```
>>> hdr['history'][0] = 'I updated this file on 2/27/09'
>>> hdr['history']
I updated this file on 2/27/09
>>> hdr['comment'][1] = 'I like using JWST observations'
>>> hdr['comment']
Edwin Hubble really knew his stuff
I like using JWST observations
```

To see the entire header as it appears in the FITS file (with the END card and padding stripped), enter the header object by itself, or `print(repr(hdr))`:

```
>>> hdr
SIMPLE  =                    T / file does conform to FITS standard
BITPIX  =                   16 / number of bits per data pixel
NAXIS   =                    0 / number of data axes
...
>>> print(repr(hdr))
SIMPLE  =                    T / file does conform to FITS standard
BITPIX  =                   16 / number of bits per data pixel
NAXIS   =                    0 / number of data axes
```

```
...
```

Entering only `print(hdr)` will also work, but may not be very legible on most displays, as this displays the header as it is written in the FITS file itself, which means there are no line breaks between cards. This is a common source of confusion for new users.

It is also possible to view a slice of the header:

```
>>> hdr[:2]
SIMPLE  =                    T / file does conform to FITS standard
BITPIX  =                   16 / number of bits per data pixel
```

Only the first two cards are shown above.

To get a list of all keywords, use the **`Header.keys()`** method just as you would with a dict:

```
>>> list(hdr.keys())
['SIMPLE', 'BITPIX', 'NAXIS', ...]
```

Examples:

See also Edit a FITS header.

Structural Keywords

FITS keywords mix up both metadata and critical information about the file structure that is needed to parse the file. These *structural* keywords are managed internally by **`astropy.io.fits`** and, in general, should not be touched by the user. Instead one should use the related attributes of the **`astropy.io.fits`** classes (see examples below).

The specific set of structural keywords used by the FITS standard varies with HDU type. The following table lists which keywords are associated with each HDU type:

Structural Keywords

HDU Type	Structural Keywords
All	`SIMPLE`, `BITPIX`, `NAXIS`
`PrimaryHDU`	`EXTEND`
`ImageHDU`, **`TableHDU`**, **`BinTableHDU`**	`PCOUNT`, `GCOUNT`
`GroupsHDU`	`NAXIS1`, `GCOUNT`, `PCOUNT`, `GROUPS`
`TableHDU`, **`BinTableHDU`**	`TFIELDS`, `TFORM`, `TBCOL`

There are many other reserved keywords, for instance for the data scaling, or for table's column attributes, as described in the FITS Standard. Most of these are accessible via attributes of the **`Column`** or HDU objects, for instance `hdu.name` to set `EXTNAME`, or `hdu.ver` for `EXTVER`. Structural keywords are checked and/or updated as a consequence of common operations. For example, when:

1. Setting the data. The `NAXIS*` keywords are set from the data shape (`.data.shape`), and `BITPIX` from the data type (`.data.dtype`).
2. Setting the header. Its keywords are updated based on the data properties (as above).
3. Writing a file. All the necessary keywords are deleted, updated or added to the header.
4. Calling an HDU's verify method (e.g., **`PrimaryHDU.verify()`**). Some keywords can be fixed automatically.

In these cases any hand-written values users might assign to those keywords will be overwrittten.

Working with Image Data

If an HDU's data is an image, the data attribute of the HDU object will return a `numpy` **`ndarray`** object. Refer to the `numpy` documentation for details on manipulating these numerical arrays:

```
>>> data = hdul[1].data
```

Here, `data` points to the data object in the second HDU (the first HDU, `hdul[0]`, being the primary HDU) which corresponds to the 'SCI' extension. Alternatively, you can access the extension by its extension name (specified in the EXTNAME keyword):

```
>>> data = hdul['SCI'].data
```

If there is more than one extension with the same EXTNAME, the EXTVER value needs to be specified along with the EXTNAME as a tuple; for example:

```
>>> data = hdul['sci',2].data
```

Note that the EXTNAME is also case-insensitive.

The returned `numpy` object has many attributes and methods for a user to get information about the array, for example:

```
>>> data.shape
(40, 40)
>>> data.dtype.name
'int16'
```

Since image data is a `numpy` object, we can slice it, view it, and perform mathematical operations on it. To see the pixel value at x=5, y=2:

```
>>> print(data[1, 4])
348
```

Note that, like C (and unlike Fortran), Python is 0-indexed and the indices have the slowest axis first and fastest changing axis last; that is, for a 2D image, the fast axis (X-axis) which corresponds to the FITS NAXIS1 keyword, is the second index. Similarly, the 1-indexed subsection of x=11 to 20 (inclusive) and y=31 to 40 (inclusive) would be given in Python as:

```
>>> data[30:40, 10:20]
array([[350, 349, 349, 348, 349, 348, 349, 347, 350, 348],
       [348, 348, 348, 349, 348, 349, 347, 348, 348, 349],
       [348, 348, 347, 349, 348, 348, 349, 349, 349, 349],
       [349, 348, 349, 349, 350, 349, 349, 347, 348, 348],
       [348, 348, 348, 348, 349, 348, 350, 349, 348, 349],
       [348, 347, 349, 349, 350, 348, 349, 348, 349, 347],
       [347, 348, 347, 348, 349, 349, 350, 349, 348, 348],
       [349, 349, 350, 348, 350, 347, 349, 349, 349, 348],
       [349, 348, 348, 348, 348, 348, 349, 347, 349, 348],
       [349, 349, 349, 348, 350, 349, 349, 350, 348, 350]],
dtype=int16)
```

To update the value of a pixel or a subsection:

```
>>> data[30:40, 10:20] = data[1, 4] = 999
```

This example changes the values of both the pixel [1, 4] and the subsection [30:40, 10:20] to the new value of 999. See the Numpy documentation for more details on Python-style array indexing and slicing.

The next example of array manipulation is to convert the image data from counts to flux:

```
>>> photflam = hdul[1].header['photflam']
>>> exptime = hdr['exptime']
>>> data = data * photflam / exptime
>>> hdul.close()
```

Note that performing an operation like this on an entire image requires holding the entire image in memory. This example performs the multiplication in-place so that no copies are made, but the original image must first be able to fit in main memory. For most observations this should not be an issue on modern personal computers.

If at this point you want to preserve all of the changes you made and write it to a new file, you can use the **`HDUList.writeto()`** method (see below).

Examples:

See also Read and plot an image from a FITS file.

Working with Table Data

This section describes reading and writing table data in the FITS format using the **`fits`** package directly. For some cases, however, the high-level Unified File Read/Write Interface will often suffice and is somewhat more convenient to use. See the Unified I/O FITS section for details.

Like images, the data portion of a FITS table extension is in the `.data` attribute:

```
>>> fits_table_filename = fits.util.get_testdata_filepath('tb.fits')
>>> hdul = fits.open(fits_table_filename)
>>> data = hdul[1].data # assuming the first extension is a table
```

If you are familiar with `numpy` **`recarray`** (record array) objects, you will find the table data is basically a record array with some extra properties. But familiarity with record arrays is not a prerequisite for this guide.

To see the first row of the table:

```
>>> print(data[0])
(1, 'abc', 3.7000000715255736, False)
```

Each row in the table is a **`FITS_record`** object which looks like a (Python) tuple containing elements of heterogeneous data types. In this example: an integer, a string, a floating point number, and a Boolean value. So the table data are just an array of such records. More commonly, a user is likely to access the data in a column-wise way. This is accomplished by using the **`field()`**

method. To get the first column (or “field” in NumPy parlance — it is used here interchangeably with “column”) of the table, use:

```
>>> data.field(0)
array([1, 2]...)
```

A `numpy` object with the data type of the specified field is returned.

Like header keywords, a column can be referred either by index, as above, or by name:

```
>>> data.field('c1')
array([1, 2]...)
```

When accessing a column by name, dict-like access is also possible (and even preferable):

```
>>> data['c1']
array([1, 2]...)
```

In most cases it is preferable to access columns by their name, as the column name is entirely independent of its physical order in the table. As with header keywords, column names are case-insensitive.

But how do we know what columns we have in a table? First, we will introduce another attribute of the table HDU: the **`columns`** attribute:

```
>>> cols = hdul[1].columns
```

This attribute is a **`ColDefs`** (column definitions) object. If we use the **`ColDefs.info()`** method from the interactive prompt:

```
>>> cols.info()
name:
    ['c1', 'c2', 'c3', 'c4']
format:
    ['1J', '3A', '1E', '1L']
unit:
    ['', '', '', '']
null:
    [-2147483647, '', '', '']
bscale:
    ['', '', 3, '']
bzero:
    ['', '', 0.4, '']
disp:
    ['I11', 'A3', 'G15.7', 'L6']
start:
    ['', '', '', '']
dim:
    ['', '', '', '']
coord_type:
    ['', '', '', '']
coord_unit:
    ['', '', '', '']
coord_ref_point:
    ['', '', '', '']
coord_ref_value:
    ['', '', '', '']
coord_inc:
    ['', '', '', '']
time_ref_pos:
    ['', '', '', '']
```

it will show the attributes of all columns in the table, such as their names, formats, bscales, bzeros, etc. A similar output that will display the column names and their formats can be printed from within a script with:

```
>>> hdul[1].columns
ColDefs(
    name = 'c1'; format = '1J'; null = -2147483647; disp = 'I11'
    name = 'c2'; format = '3A'; disp = 'A3'
    name = 'c3'; format = '1E'; bscale = 3; bzero = 0.4; disp =
'G15.7'
    name = 'c4'; format = '1L'; disp = 'L6'
)
```

We can also get these properties individually; for example:

```
>>> cols.names
['c1', 'c2', 'c3', 'c4']
```

returns a (Python) list of field names.

Since each field is a `numpy` object, we will have the entire arsenal of `numpy` tools to use. We can reassign (update) the values:

```
>>> data['c4'][:] = 0
```

take the mean of a column:

```
>>> data['c3'].mean()
5.19999989271164
```

and so on.

Examples:

See also Accessing data stored as a table in a multi-extension FITS (MEF) file.

Save File Changes

As mentioned earlier, after a user opened a file, made a few changes to either header or data, the user can use `HDUList.writeto()` to save the changes. This takes the version of headers and data in memory and writes them to a new FITS file on disk. Subsequent operations can be performed to the data in memory and written out to yet another different file, all without recopying the original data to (more) memory:

```
hdul.writeto('newtable.fits')
```

will write the current content of `hdulist` to a new disk file newfile.fits. If a file was opened with the update mode, the `HDUList.flush()` method can also be used to write all of the changes made since `open()`, back to the original file. The `close()` method will do the same for a FITS file opened with update mode:

```
with fits.open('original.fits', mode='update') as hdul:
    # Change something in hdul.
    hdul.flush()  # changes are written back to original.fits

# closing the file will also flush any changes and prevent further
writing
```

Creating a New FITS File

Creating a New Image File

So far we have demonstrated how to read and update an existing FITS file. But how about creating a new FITS file from scratch? Such tasks are very convenient in `astropy` for an image HDU. We will first demonstrate how to create a FITS file consisting of only the primary HDU with image data.

First, we create a `numpy` object for the data part:

```
>>> import numpy as np
>>> n = np.arange(100.0) # a simple sequence of floats from 0.0 to
99.9
```

Next, we create a **`PrimaryHDU`** object to encapsulate the data:

```
>>> hdu = fits.PrimaryHDU(n)
```

We then create an **`HDUList`** to contain the newly created primary HDU, and write to a new file:

```
>>> hdul = fits.HDUList([hdu])
>>> hdul.writeto('new1.fits')
```

That is it! In fact, `astropy` even provides a shortcut for the last two lines to accomplish the same behavior:

```
>>> hdu.writeto('new2.fits')
```

This will write a single HDU to a FITS file without having to manually encapsulate it in an **`HDUList`** object first.

Creating a New Table File

Note

If you want to create a **binary** FITS table with no other HDUs, you can use **`Table`** instead and then write to FITS. This is less complicated than “lower-level” FITS interface:

```
>>> from astropy.table import Table
>>> t = Table([[1, 2], [4, 5], [7, 8]], names=('a', 'b', 'c'))
>>> t.write('table1.fits', format='fits')
```

The equivalent code using `astropy.io.fits` would look like this:

```
>>> from astropy.io import fits
>>> import numpy as np
>>> c1 = fits.Column(name='a', array=np.array([1, 2]),
format='K')
>>> c2 = fits.Column(name='b', array=np.array([4, 5]),
format='K')
>>> c3 = fits.Column(name='c', array=np.array([7, 8]),
format='K')
>>> t = fits.BinTableHDU.from_columns([c1, c2, c3])
>>> t.writeto('table2.fits')
```

To create a table HDU is a little more involved than an image HDU, because a table's structure needs more information. First of all, tables can only be an extension HDU, not a primary. There are two kinds of FITS table extensions: ASCII and binary. We will use binary table examples here.

To create a table from scratch, we need to define columns first, by constructing the `Column` objects and their data. Suppose we have two columns, the first containing strings, and the second containing floating point numbers:

```
>>> import numpy as np
>>> a1 = np.array(['NGC1001', 'NGC1002', 'NGC1003'])
>>> a2 = np.array([11.1, 12.3, 15.2])
>>> col1 = fits.Column(name='target', format='20A', array=a1)
>>> col2 = fits.Column(name='V_mag', format='E', array=a2)
```

Note

It is not necessary to create a `Column` object explicitly if the data is stored in a structured array.

Next, create a `ColDefs` (column-definitions) object for all columns:

```
>>> cols = fits.ColDefs([col1, col2])
```

Now, create a new binary table HDU object by using the `BinTableHDU.from_columns()` function:

```
>>> hdu = fits.BinTableHDU.from_columns(cols)
```

This function returns (in this case) a `BinTableHDU`.

The data structure used to represent FITS tables is called a `FITS_rec` and is derived from the `numpy.recarray` interface. When creating a new table HDU the individual column arrays will be assembled into a single `FITS_rec` array.

You can create a **BinTableHDU** more concisely without creating intermediate variables for the individual columns and without manually creating a **ColDefs** object:

```
>>> hdu = fits.BinTableHDU.from_columns(
...     [fits.Column(name='target', format='20A', array=a1),
...      fits.Column(name='V_mag', format='E', array=a2)])
```

Now you may write this new table HDU directly to a FITS file like so:

```
>>> hdu.writeto('table3.fits')
```

This shortcut will automatically create a minimal primary HDU with no data and prepend it to the table HDU to create a valid FITS file. If you require additional data or header keywords in the primary HDU you may still create a **PrimaryHDU** object and build up the FITS file manually using an **HDUList**, as described in the next section.

Creating a File with Multiple Extensions

In the previous examples we created files with a single meaningful extension (a **PrimaryHDU** or **BinTableHDU**). To create a file with multiple extensions we need to create extension HDUs and append them to an **HDUList**.

First, we create some data for Image extensions:

```
>>> import numpy as np
>>> n = np.ones((3, 3))
>>> n2 = np.ones((100, 100))
>>> n3 = np.ones((10, 10, 10))
```

Note that the data shapes of the different extensions do not need to be the same. Next, place the data into separate **PrimaryHDU** and **ImageHDU** objects:

```
>>> primary_hdu = fits.PrimaryHDU(n)
>>> image_hdu = fits.ImageHDU(n2)
>>> image_hdu2 = fits.ImageHDU(n3)
```

A multi-extension FITS file is not constrained to be only imaging or table data, we can mix them. To show this we'll use the example from the previous section to make a **BinTableHDU**:

```
>>> c1 = fits.Column(name='a', array=np.array([1, 2]), format='K')
>>> c2 = fits.Column(name='b', array=np.array([4, 5]), format='K')
>>> c3 = fits.Column(name='c', array=np.array([7, 8]), format='K')
>>> table_hdu = fits.BinTableHDU.from_columns([c1, c2, c3])
```

Now when we create the `HDUList` we list all extensions we want to include:

```
>>> hdul = fits.HDUList([primary_hdu, image_hdu, table_hdu])
```

Because `HDUList` acts like a `list` we can also append, for example, an `ImageHDU` to an already existing `HDUList`:

```
>>> hdul.append(image_hdu2)
```

Multi-extension `HDUList` are treated just like those with only a `PrimaryHDU`, so to save the file use `HDUList.writeto()` as shown above.

Note

The FITS standard enforces all files to have exactly one `PrimaryHDU` that is the first HDU present in the file. This standard is enforced during the call to `HDUList.writeto()` and an error will be raised if it is not met. See the `output_verify` option in `HDUList.writeto()` for ways to fix or ignore these warnings.

In the previous example the `PrimaryHDU` contained actual data. In some cases it is desirable to have a minimal `PrimaryHDU` with only basic header information. To do this, first create a new `Header` object to encapsulate any keywords you want to include in the primary HDU, then as before create a `PrimaryHDU`:

```
>>> hdr = fits.Header()
>>> hdr['OBSERVER'] = 'Edwin Hubble'
>>> hdr['COMMENT'] = "Here's some commentary about this FITS file."
>>> empty_primary = fits.PrimaryHDU(header=hdr)
```

When we create a new primary HDU with a custom header as in the above example, this will automatically include any additional header keywords that are *required* by the FITS format (keywords such as `SIMPLE` and `NAXIS` for example). In general, users should not have to manually manage such keywords, and should only create and modify observation-specific informational keywords.

We then create an HDUList containing both the primary HDU and any other HDUs want:

```
>>> hdul = fits.HDUList([empty_primary, image_hdu2, table_hdu])
```

Examples:

See also Create a multi-extension FITS (MEF) file from scratch.

Convenience Functions

`astropy.io.fits` also provides several high-level ("convenience") functions. Such a convenience function is a "canned" operation to achieve one task. By using these "convenience" functions, a user does not have to worry about opening or closing a file; all of the housekeeping is done implicitly.

Warning

These functions are useful for interactive Python sessions and less complex analysis scripts, but should not be used for application code, as they are highly inefficient. For example, each call to `getval()` requires re-parsing the entire FITS file. Code that makes repeated use of these functions should instead open the file with `open()` and access the data structures directly.

The first of these functions is `getheader()`, to get the header of an HDU. Here are several examples of getting the header. Only the file name is required for this function. The rest of the arguments are optional and flexible to specify which HDU the user wants to access:

```
>>> from astropy.io.fits import getheader
>>> hdr = getheader(fits_image_filename)  # get default HDU (=0),
i.e. primary HDU's header
>>> hdr = getheader(fits_image_filename, 0)  # get primary HDU's
header
>>> hdr = getheader(fits_image_filename, 2)  # the second extension
>>> hdr = getheader(fits_image_filename, 'sci')  # the first HDU with
EXTNAME='SCI'
>>> hdr = getheader(fits_image_filename, 'sci', 2)  # HDU with
EXTNAME='SCI' and EXTVER=2
>>> hdr = getheader(fits_image_filename, ('sci', 2))  # use a tuple
to do the same
>>> hdr = getheader(fits_image_filename, ext=2)  # the second
extension
>>> hdr = getheader(fits_image_filename, extname='sci')  # first HDU
with EXTNAME='SCI'
>>> hdr = getheader(fits_image_filename, extname='sci', extver=2)
```

Ambiguous specifications will raise an exception:

```
>>> getheader(fits_image_filename, ext=('sci', 1), extname='err',
extver=2)
Traceback (most recent call last):
    ...
TypeError: Redundant/conflicting extension arguments(s): ...
```

After you get the header, you can access the information in it, such as getting and modifying a keyword value:

```
>>> fits_image_2_filename =
fits.util.get_testdata_filepath('o4sp040b0_raw.fits')
>>> hdr = getheader(fits_image_2_filename, 0)    # get primary hdu's
header
>>> filter = hdr['filter']                       # get the value of
the keyword "filter'
>>> val = hdr[10]                                # get the 11th
keyword's value
>>> hdr['filter'] = 'FW555'                      # change the keyword
value
```

For the header keywords, the header is like a dictionary, as well as a list. The user can access the keywords either by name or by numeric index, as explained earlier in this chapter.

If a user only needs to read one keyword, the **`getval()`** function can further simplify to just one call, instead of two as shown in the above examples:

```
>>> from astropy.io.fits import getval
>>> # get 0th extension's keyword FILTER's value
>>> flt = getval(fits_image_2_filename, 'filter', 0)
>>> flt
'Clear'

>>> # get the 2nd sci extension's 11th keyword's value
>>> val = getval(fits_image_2_filename, 10, 'sci', 2)
>>> val
False
```

The function **`getdata()`** gets the data of an HDU. Similar to **`getheader()`**, it only requires the input FITS file name while the extension is specified through the optional arguments. It does have one extra optional argument header. If header is set to True, this function will return both data and header, otherwise only data is returned:

```
>>> from astropy.io.fits import getdata
>>> # get 3rd sci extension's data:
```

```
>>> data = getdata(fits_image_filename, 'sci', 3)
>>> # get 1st extension's data AND header:
>>> data, hdr = getdata(fits_image_filename, 1, header=True)
```

The functions introduced above are for reading. The next few functions demonstrate convenience functions for writing:

```
>>> fits.writeto('out.fits', data, hdr)
```

The **writeto()** function uses the provided data and an optional header to write to an output FITS file.

```
>>> fits.append('out.fits', data, hdr)
```

The **append()** function will use the provided data and the optional header to append to an existing FITS file. If the specified output file does not exist, it will create one.

```
from astropy.io.fits import update
update(filename, dat, hdr, 'sci')         # update the 'sci'
extension
update(filename, dat, 3)                  # update the 3rd extension
update(filename, dat, hdr, 3)             # update the 3rd extension
update(filename, dat, 'sci', 2)           # update the 2nd SCI
extension
update(filename, dat, 3, header=hdr)      # update the 3rd extension
update(filename, dat, header=hdr, ext=5)  # update the 5th extension
```

The **update()** function will update the specified extension with the input data/header. The third argument can be the header associated with the data. If the third argument is not a header, it (and other positional arguments) are assumed to be the extension specification(s). Header and extension specs can also be keyword arguments.

The **printdiff()** function will print a difference report of two FITS files, including headers and data. The first two arguments must be two FITS filenames or FITS file objects with matching data types (i.e., if using strings to specify filenames, both inputs must be strings). The third argument is an optional extension specification, with the same call format of **getheader()** and **getdata()**. In addition you can add any keywords accepted by the **FITSDiff** class.

```
from astropy.io.fits import printdiff
# get a difference report of ext 2 of inA and inB
printdiff('inA.fits', 'inB.fits', ext=2)
```

```
# ignore HISTORY and COMMMENT keywords
printdiff('inA.fits', 'inB.fits', ignore_keywords=
('HISTORY','COMMENT')
```

Finally, the **info()** function will print out information of the specified FITS file:

```
>>> fits.info(fits_image_filename)
Filename: ...test0.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU     138   ()
  1  SCI           1 ImageHDU        61   (40, 40)   int16
  2  SCI           2 ImageHDU        61   (40, 40)   int16
  3  SCI           3 ImageHDU        61   (40, 40)   int16
  4  SCI           4 ImageHDU        61   (40, 40)   int16
```

This is one of the most useful convenience functions for getting an overview of what a given file contains without looking at any of the details.

Using astropy.io.fits

FITS Headers

In the next three chapters, more detailed information including examples will be explained for manipulating FITS headers, image/array data, and table data respectively.

Header of an HDU

Every Header Data Unit (HDU) normally has two components: header and data. In `astropy` these two components are accessed through the two attributes of the HDU, `hdu.header` and `hdu.data`.

While an HDU may have empty data (i.e., the `.data` attribute is **None**), any HDU will always have a header. When an HDU is created with a constructor (e.g., `hdu = PrimaryHDU(data, header)`), the user may supply the header value from an existing HDU's header and the data value from a `numpy` array. If the defaults (None) are used, the new HDU will have the minimal required keywords for an HDU of that type:

```
>>> from astropy.io import fits
>>> hdu = fits.PrimaryHDU()
>>> hdu.header  # show the all of the header cards
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                    8 / array data type
```

```
NAXIS   =                    0 / number of array dimensions
EXTEND  =                    T
```

A user can use any header and any data to construct a new HDU. `astropy` will strip any keywords that describe the data structure leaving only your informational keywords. Later it will add back in the required structural keywords for compatibility with the new HDU and any data added to it. So, a user can use a table HDU's header to construct an image HDU and vice versa. The constructor will also ensure the data type and dimension information in the header agree with the data.

The Header Attribute

Value Access, Updating, and Creating

As shown in the Getting Started tutorial, keyword values can be accessed via keyword name or index of an HDU's header attribute. You can also use the wildcard character `*` to get the keyword value pairs that match your search string. Here is a quick summary:

```
>>> fits_image_filename =
fits.util.get_testdata_filepath('test0.fits')
>>> hdul = fits.open(fits_image_filename)  # open a FITS file
>>> hdr = hdul[0].header  # the primary HDU header
>>> print(hdr[34])  # get the 2nd keyword's value
96
>>> hdr[34] = 20  # change its value
>>> hdr['DARKCORR']  # get the value of the keyword 'darkcorr'
'OMIT'
>>> hdr['DARKCOR*']  # get keyword values using wildcard matching
DARKCORR= 'OMIT              ' / Do dark correction: PERFORM, OMIT,
COMPLETE
>>> hdr['darkcorr'] = 'PERFORM'  # change darkcorr's value
```

Keyword names are case-insensitive except in a few special cases (see the sections on HIERARCH card and record-valued cards). Thus, `hdr['abc']`, `hdr['ABC']`, or `hdr['aBc']` are all equivalent.

As with Python's **`dict`** type, new keywords can also be added to the header using assignment syntax:

```
>>> hdr = hdul[1].header
>>> 'DARKCORR' in hdr  # Check for existence
False
```

```
>>> hdr['DARKCORR'] = 'OMIT'  # Add a new DARKCORR keyword
```

You can also add a new value *and* comment by assigning them as a tuple:

```
>>> hdr['DARKCORR'] = ('OMIT', 'Dark Image Subtraction')
```

Note

An important point to note when adding new keywords to a header is that by default they are not appended *immediately* to the end of the file. Rather, they are appended to the last non-commentary keyword. This is in order to support the common use case of always having all HISTORY keywords grouped together at the end of a header. A new non-commentary keyword will be added at the end of the existing keywords, but before any HISTORY/COMMENT keywords at the end of the header.

There are a couple of ways to override this functionality:

- Use the **`Header.append()`** method with the `end=True` argument:

  ```
  >>> hdr.append(('DARKCORR', 'OMIT', 'Dark Image
  Subtraction'), end=True)
  ```

  This forces the new keyword to be added at the actual end of the header.

- The **`Header.insert()`** method will always insert a new keyword exactly where you ask for it:

  ```
  >>> del hdr['DARKCORR']  # Delete previous insertion for
  doctest
  >>> hdr.insert(20, ('DARKCORR', 'OMIT', 'Dark Image
  Subtraction'))
  ```

  This inserts the DARKCORR keyword before the 20th keyword in the header no matter what it is.

A keyword (and its corresponding card) can be deleted using the same index/name syntax:

```
>>> del hdr[3]  # delete the 2nd keyword
>>> del hdr['DARKCORR']  # delete the value of the keyword 'DARKCORR'
```

Note that, like a regular Python list, the indexing updates after each delete, so if `del hdr[3]` is done two times in a row, the fourth and fifth keywords are removed from the original header. Likewise, `del hdr[-1]` will delete the last card in the header.

It is also possible to delete an entire range of cards using the slice syntax:

```
>>> del hdr[3:5]
```

The method **`Header.set()`** is another way to update the value or comment associated with an existing keyword, or to create a new keyword. Most of its functionality can be duplicated with the dict-like syntax shown above. But in some cases it might be more clear. It also has the advantage of allowing a user to either move cards within the header or specify the location of a new card relative to existing cards:

```
>>> hdr.set('target', 'NGC1234', 'target name')
>>> # place the next new keyword before the 'TARGET' keyword
>>> hdr.set('newkey', 666, before='TARGET')  # comment is optional
>>> # place the next new keyword after the 21st keyword
>>> hdr.set('newkey2', 42.0, 'another new key', after=20)
```

In FITS headers, each keyword may also have a comment associated with it explaining its purpose. The comments associated with each keyword are accessed through the **`comments`** attribute:

```
>>> hdr['NAXIS']
2
>>> hdr.comments['NAXIS']
'number of data axes'
>>> hdr.comments['NAXIS'] = 'The number of image axes'  # Update
>>> hdul.close()  # close the HDUList again
```

Comments can be accessed in all of the same ways that values are accessed, whether by keyword name or card index. Slices are also possible. The only difference is that you go through `hdr.comments` instead of just `hdr` by itself.

COMMENT, HISTORY, and Blank Keywords

Most keywords in a FITS header have unique names. If there are more than two cards sharing the same name, it is the first one accessed when referred by name. The duplicates can only be accessed by numeric indexing.

There are three special keywords (their associated cards are sometimes referred to as commentary cards), which commonly appear in FITS headers more than once. They are (1) blank keyword, (2) HISTORY, and (3) COMMENT. Unlike other keywords, when accessing these keywords they are returned as a list:

```
>>> filename = fits.util.get_testdata_filepath('history_header.fits')
>>> with fits.open(filename) as hdul:  # open a FITS file
...     hdr = hdul[0].header

>>> hdr['HISTORY']
I updated this file on 02/03/2011
I updated this file on 02/04/2011
```

These lists can be sliced like any other list. For example, to display just the last HISTORY entry, use `hdr['history'][-1]`. Existing commentary cards can also be updated by using the appropriate index number for that card.

New commentary cards can be added like any other card by using the dict-like keyword assignment syntax, or by using the **`Header.set()`** method. However, unlike with other keywords, a new commentary card is always added and appended to the last commentary card with the same keyword, rather than to the end of the header.

Example

To add a new commentary card:

```
>>> hdu.header['HISTORY'] = 'history 1'
>>> hdu.header[''] = 'blank 1'
>>> hdu.header['COMMENT'] = 'comment 1'
>>> hdu.header['HISTORY'] = 'history 2'
>>> hdu.header[''] = 'blank 2'
>>> hdu.header['COMMENT'] = 'comment 2'
```

and the part in the modified header becomes:

```
HISTORY history 1
HISTORY history 2
        blank 1
        blank 2
COMMENT comment 1
COMMENT comment 2
```

Users can also directly control exactly where in the header to add a new commentary card by using the **`Header.insert()`** method.

Note

Ironically, there is no comment in a commentary card, only a string value.

Undefined Values

FITS headers can have undefined values and these are represented in Python with the special value **None**. **None** can be used when assigning values to a **Header** or **Card**.

```
>>> hdr = fits.Header()
>>> hdr['UNDEF'] = None
>>> hdr['UNDEF'] is None
True
>>> repr(hdr)
'UNDEF   =
'
>>> hdr.append('UNDEF2')
>>> hdr['UNDEF2'] is None
True
>>> hdr.append(('UNDEF3', None, 'Undefined value'))
>>> str(hdr.cards[-1])
'UNDEF3  =  / Undefined value
'
```

Card Images

A FITS header consists of card images.

A card image in a FITS header consists of a keyword name, a value, and optionally a comment. Physically, it takes 80 columns (bytes) — without carriage return — in a FITS file's storage format. In `astropy`, each card image is manifested by a **Card** object. There are also special kinds of cards: commentary cards (see above) and card images taking more than one 80-column card image. The latter will be discussed later.

Most of the time the details of dealing with cards are handled by the **Header** object, and it is not necessary to directly manipulate cards. In fact, most **Header** methods that accept a `(keyword, value)` or `(keyword, value, comment)` tuple as an argument can also take a **Card** object as an argument. **Card** objects are just wrappers around such tuples that provide the logic for parsing and formatting individual cards in a header. There is usually nothing gained by manually using a **Card** object, except to examine how a card might appear in a header before actually adding it to the header.

A new Card object is created with the **Card** constructor: `Card(key, value, comment)`.

Example
To create a new Card object:

```
>>> c1 = fits.Card('TEMP', 80.0, 'temperature, floating value')
>>> c2 = fits.Card('DETECTOR', 1)  # comment is optional
>>> c3 = fits.Card('MIR_REVR', True,
...                'mirror reversed? Boolean value')
>>> c4 = fits.Card('ABC', 2+3j, 'complex value')
>>> c5 = fits.Card('OBSERVER', 'Hubble', 'string value')

>>> print(c1); print(c2); print(c3); print(c4); print(c5)  # show the
cards
TEMP    =                 80.0 / temperature, floating value
DETECTOR=                    1
MIR_REVR=                    T / mirror reversed? Boolean value
ABC     =           (2.0, 3.0) / complex value
OBSERVER= 'Hubble  '           / string value
```

Cards have the attributes `.keyword`, `.value`, and `.comment`. Both `.value` and `.comment` can be changed but not the `.keyword` attribute. In other words, once a card is created, it is created for a specific, immutable keyword.

The **`Card()`** constructor will check if the arguments given are conforming to the FITS standard and has a fixed card image format. If the user wants to create a card with a customized format or even a card which is not conforming to the FITS standard (e.g., for testing purposes), the **`Card.fromstring()`** class method can be used.

Cards can be verified with **`Card.verify()`**. The nonstandard card `c2` in the example below is flagged by such verification. More about verification in `astropy` will be discussed in a later chapter.

```
>>> c1 = fits.Card.fromstring('ABC     = 3.456D023')
>>> c2 = fits.Card.fromstring("P.I. ='Hubble'")
>>> print(c1)
ABC     = 3.456D023
>>> print(c2)
P.I. ='Hubble'
>>> c2.verify()
Output verification result:
Unfixable error: Illegal keyword name 'P.I.'
```

A list of the **`Card`** objects underlying a **`Header`** object can be accessed with the **`Header.cards`** attribute. This list is only meant for observing, and should not be directly manipulated. In fact, it is only a copy — modifications to it will not affect the header from which it came. Use the methods provided by the **`Header`** class instead.

CONTINUE Cards

The fact that the FITS standard only allows up to eight characters for the keyword name and 80 characters to contain the keyword, the value, and the comment is restrictive for certain applications. To allow long string values for keywords, a proposal was made in:

> https://heasarc.gsfc.nasa.gov/docs/heasarc/ofwg/docs/ofwg_recomm/r13.html

by using the CONTINUE keyword after the regular 80 column containing the keyword. `astropy` does support this convention, which is a part of the FITS standard since version 4.0.

Examples

The examples below show that the use of CONTINUE is automatic for long string values:

```
>>> hdr = fits.Header()
>>> hdr['abc'] = 'abcdefg' * 20
>>> hdr
ABC     =
'abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcd&
'
CONTINUE
'efgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefga&
'
CONTINUE  'bcdefg'
>>> hdr['abc']
'abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcde
fgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcd
efg'
>>> # both value and comments are long
>>> hdr['abc'] = ('abcdefg' * 10, 'abcdefg' * 10)
>>> hdr
ABC     =
'abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcd&
'
CONTINUE  'efg&'
CONTINUE  '&' /
abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefga
CONTINUE  '' / bcdefg
```

Note that when a CONTINUE card is used, at the end of each 80-character card image, an ampersand is present. The ampersand is not part of the string value. Also, there is no “=” at the ninth column after CONTINUE. In the first example, the entire 240 characters is treated by `astropy` as a single card.

So, if it is the nth card in a header, the (n+1)th card refers to the next keyword, not the next CONTINUE card. As such, CONTINUE cards are transparently handled by `astropy` as a single logical card, and it is generally not necessary to worry about the details of the format. Keywords that resolve to a set of CONTINUE cards can be accessed and updated just like regular keywords.

HIERARCH Cards

For keywords longer than eight characters, there is a convention originated at the European Southern Observatory (ESO) to facilitate such use. It uses a special keyword HIERARCH with the actual long keyword following. `astropy` supports this convention as well.

If a keyword contains more than eight characters `astropy` will automatically use a HIERARCH card, but will also issue a warning in case this is in error. However, you may explicitly request a HIERARCH card by prepending the keyword with 'HIERARCH ' (just as it would appear in the header). For example, `hdr['HIERARCH abcdefghi']` will create the keyword `abcdefghi` without displaying a warning. Once created, HIERARCH keywords can be accessed like any other: `hdr['abcdefghi']`, without prepending 'HIERARCH' to the keyword.

Examples

`astropy` will use a HIERARCH card and issue a warning when keywords contain more than eight characters:

```
>>> # this will result in a Warning because a HIERARCH card is
implicitly created
>>> c = fits.Card('abcdefghi', 10)
>>> print(c)
HIERARCH abcdefghi = 10
>>> c = fits.Card('hierarch abcdefghi', 10)
>>> print(c)
HIERARCH abcdefghi = 10
>>> hdu = fits.PrimaryHDU()
>>> hdu.header['hierarch abcdefghi'] =  99
>>> hdu.header['abcdefghi']
99
>>> hdu.header['abcdefghi'] = 10
>>> hdu.header['abcdefghi']
10
>>> hdu.header
```

```
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                    8 / array data type
NAXIS   =                    0 / number of array dimensions
EXTEND  =                    T
HIERARCH abcdefghi = 10
```

Note

A final point to keep in mind about the **Header** class is that much of its design is intended to abstract away quirks about the FITS format. This is why, for example, it will automatically create CONTINUE and HIERARCH cards. The Header is just a data structure, and as a user you should not have to worry about how it ultimately gets serialized to a header in a FITS file.

Though there are some areas where it is almost impossible to hide away the quirks of the FITS format, `astropy` tries to make it so that you have to think about it as little as possible. If there are any areas that are left vague or difficult to understand about how the header is constructed, please let us know, as there are probably areas where this can be improved on even more.

Image Data

In this chapter, we will discuss the data component in an image HDU.

Image Data as an Array

A FITS primary HDU or an image extension HDU may contain image data. The following discussions apply to both of these HDU classes. For most cases in `astropy`, it is a `numpy` array, having the shape specified by the NAXIS keywords and the data type specified by the BITPIX keyword — unless the data is scaled, in which case see the next section. Here is a quick cross reference between allowed BITPIX values in FITS images and the `numpy` data types:

```
BITPIX    Numpy Data Type
8         numpy.uint8 (note it is UNsigned integer)
16        numpy.int16
32        numpy.int32
64        numpy.int64
-32       numpy.float32
-64       numpy.float64
```

To recap, in `numpy` the arrays are 0-indexed and the axes are ordered from slow to fast. So, if a FITS image has NAXIS1=300 and NAXIS2=400, the `numpy` array of its data will have the shape of (400, 300).

Examples

Here is a summary of reading and updating image data values:

```
>>> from astropy.io import fits
>>> fits_image_filename =
fits.util.get_testdata_filepath('test0.fits')

>>> with fits.open(fits_image_filename) as hdul:  # open a FITS file
...     data = hdul[1].data  # assume the first extension is an image
>>> print(data[1, 4])   # get the pixel value at x=5, y=2
313
>>> # get values of the subsection from x=11 to 20, y=31 to 40
(inclusive)
>>> data[30:40, 10:20]
array([[314, 314, 313, 312, 313, 313, 313, 313, 313, 312],
       [314, 314, 312, 313, 313, 311, 313, 312, 312, 314],
       [314, 315, 313, 313, 313, 313, 315, 312, 314, 312],
       [314, 313, 313, 314, 311, 313, 313, 313, 313, 313],
       [313, 314, 312, 314, 312, 314, 314, 315, 313, 313],
       [312, 311, 311, 312, 312, 312, 312, 313, 311, 312],
       [314, 314, 314, 314, 312, 313, 314, 314, 314, 311],
       [314, 313, 312, 313, 313, 314, 312, 312, 311, 314],
       [313, 313, 313, 314, 313, 313, 315, 313, 312, 313],
       [314, 313, 313, 314, 313, 312, 312, 314, 310, 314]],
dtype=int16)
>>> data[1,4] = 999  # update a pixel value
>>> data[30:40, 10:20] = 0  # update values of a subsection
>>> data[3] = data[2]    # copy the 3rd row to the 4th row
```

Here are some more complicated examples by using the concept of the “mask array.” The first example is to change all negative pixel values in `data` to zero. The second one is to take logarithm of the pixel values which are positive:

```
>>> data[data < 0] = 0
>>> import numpy as np
>>> data[data > 0] = np.log(data[data > 0])
```

These examples show the concise nature of `numpy` array operations.

Scaled Data

Sometimes an image is scaled; that is, the data stored in the file is not the image's physical (true) values, but linearly transformed according to the equation:

```
physical value = BSCALE * (storage value) + BZERO
```

BSCALE and BZERO are stored as keywords of the same names in the header of the same HDU. The most common use of a scaled image is to store unsigned 16-bit integer data because the FITS standard does not allow it. In this case, the stored data is signed 16-bit integer (BITPIX=16) with BZERO=32768 (\(2^{15}\)), BSCALE=1.

Reading Scaled Image Data

Images are scaled only when either of the BSCALE/BZERO keywords are present in the header and either of their values is not the default value (BSCALE=1, BZERO=0).

For unscaled data, the data attribute of an HDU in `astropy` is a `numpy` array of the same data type specified by the BITPIX keyword. For a scaled image, the `.data` attribute will be the physical data (i.e., already transformed from the storage data and may not be the same data type as prescribed in BITPIX). This means an extra step of copying is needed and thus the corresponding memory requirement. This also means that the advantage of memory mapping is reduced for scaled data.

For floating point storage data, the scaled data will have the same data type. For integer data type, the scaled data will always be single precision floating point (`numpy.float32`).

Example

Here is an example of what happens to scaled data, before and after the data is touched:

```
>>> fits_scaledimage_filename =
fits.util.get_testdata_filepath('scale.fits')

>>> hdul = fits.open(fits_scaledimage_filename)
>>> hdu = hdul[0]
>>> hdu.header['bitpix']
16
>>> hdu.header['bzero']
1500.0
>>> hdu.data[0, 0]  # once data is touched, it is scaled
557.7563
>>> hdu.data.dtype.name
'float32'
>>> hdu.header['bitpix']  # BITPIX is also updated
```

```
-32
>>> # BZERO and BSCALE are removed after the scaling
>>> hdu.header['bzero']
Traceback (most recent call last):
    ...
KeyError: "Keyword 'BZERO' not found."
```

Warning

An important caveat to be aware of when dealing with scaled data in `astropy`, is that when accessing the data via the `.data` attribute, the data is automatically scaled with the BZERO and BSCALE parameters. If the file was opened in "update" mode, it will be saved with the rescaled data. This surprising behavior is a compromise to err on the side of not losing data: if some floating point calculations were made on the data, rescaling it when saving could result in a loss of information.

To prevent this automatic scaling, open the file with the `do_not_scale_image_data=True` argument to `fits.open()`. This is especially useful for updating some header values, while ensuring that the data is not modified.

You may also manually reapply scale parameters by using `hdu.scale()` (see below). Alternately, you may open files with the `scale_back=True` argument. This assures that the original scaling is preserved when saving even when the physical values are updated. In other words, it reapplies the scaling to the new physical values upon saving.

Writing Scaled Image Data

With the extra processing and memory requirement, we discourage the use of scaled data as much as possible. However, `astropy` does provide ways to write scaled data with the **`scale`** method.

Examples

To write scaled data with the **`scale`** method:

```
>>> # scale the data to Int16 with user specified bscale/bzero
>>> hdu.scale('int16', bzero=32768)
>>> # scale the data to Int32 with the min/max of the data range,
emits
>>> # RuntimeWarning: overflow encountered in short_scalars
>>> hdu.scale('int32', 'minmax')
>>> # scale the data, using the original BSCALE/BZERO, emits
>>> # RuntimeWarning: invalid value encountered in add
>>> hdu.scale('int32', 'old')
>>> hdul.close()
```

The first example above shows how to store an unsigned short integer array.

Caution must be exercised when using the **scale()** method. The **data** attribute of an image HDU, after the **scale()** call, will become the storage values, not the physical values. So, only call **scale()** just before writing out to FITS files (i.e., calls of **writeto()**, **flush()**, or **close()**). No further use of the data should be exercised. Here is an example of what happens to the **data** attribute after the **scale()** call:

```
>>> hdu = fits.PrimaryHDU(np.array([0., 1, 2, 3]))
>>> print(hdu.data)
[0. 1. 2. 3.]
>>> hdu.scale('int16', bzero=32768)
>>> print(hdu.data)  # now the data has storage values
[-32768 -32767 -32766 -32765]
>>> hdu.writeto('new.fits')
```

Data Sections

When a FITS image HDU's **data** is accessed, either the whole data is copied into memory (in cases of NOT using memory mapping or if the data is scaled) or a virtual memory space equivalent to the data size is allocated (in the case of memory mapping of non-scaled data). If there are several very large image HDUs being accessed at the same time, the system may run out of memory.

If a user does not need the entire image(s) at the same time (e.g., processing the images(s) ten rows at a time), the **section** attribute of an HDU can be used to alleviate such memory problems.

With `astropy`'s improved support for memory-mapping, the sections feature is not as necessary as it used to be for handling very large images. However, if the image's data is scaled with non-trivial BSCALE/BZERO values, accessing the data in sections may still be necessary under the current implementation. Memmap is also insufficient for loading images larger than 2 to 4 GB on a 32-bit system — in such cases it may be necessary to use sections.

Example

Here is an example of getting the median image from three input images of the size 5000x5000.

```
hdul1 = fits.open('file1.fits')
hdul2 = fits.open('file2.fits')
hdul3 = fits.open('file3.fits')
output = np.zeros((5000, 5000))
for i in range(50):
    j = i * 100
    k = j + 100
    x1 = hdul1[0].section[j:k,:]
    x2 = hdul2[0].section[j:k,:]
    x3 = hdul3[0].section[j:k,:]
    output[j:k, :] = np.median([x1, x2, x3], axis=0)
```

Data in each `section` does not need to be contiguous for memory savings to be possible. `astropy` will do its best to join together discontiguous sections of the array while reading as little as possible into main memory.

Sections cannot currently be assigned. Any modifications made to a data section are not saved back to the original file.

Table Data

In this chapter, we will discuss the data component in a table HDU. A table will always be in an extension HDU, never in a primary HDU.

There are two kinds of tables in the FITS standard: binary tables and ASCII tables. Binary tables are more economical in storage and faster in data access and manipulation. ASCII tables store the data in a “human readable” form and therefore take up more storage space as well as more processing time since the ASCII text needs to be parsed into numerical values.

Note

If you want to read or write a single table in FITS format then the most convenient method is often via the high-level Unified File Read/Write Interface. In particular see the Unified I/O FITS section.

Table Data as a Record Array

What is a Record Array?

A record array is an array which contains records (i.e., rows) of heterogeneous data types. Record arrays are available through the records module in the NumPy library.

Here is a sample record array:

```
>>> import numpy as np
>>> bright = np.rec.array([(1,'Sirius', -1.45, 'A1V'),
...                        (2,'Canopus', -0.73, 'F0Ib'),
...                        (3,'Rigil Kent', -0.1, 'G2V')],
...                       formats='int16,a20,float32,a10',
...                       names='order,name,mag,Sp')
```

In this example, there are three records (rows) and four fields (columns). The first field is a short integer, the second a character string (of length 20), the third a floating point number, and the fourth a character string (of length 10). Each record has the same (heterogeneous) data structure.

The underlying data structure used for FITS tables is a class called `FITS_rec` which is a specialized subclass of `numpy.recarray`. A `FITS_rec` can be instantiated directly using the same initialization format presented for plain recarrays as in the example above. You may also instantiate a new `FITS_rec` from a list of `astropy.io.fits.Column` objects using the `FITS_rec.from_columns()` class method. This has the exact same semantics as `BinTableHDU.from_columns()` and `TableHDU.from_columns()`, except that it only returns an actual FITS_rec array and not a whole HDU object.

Metadata of a Table

The data in a FITS table HDU is basically a record array with added attributes. The metadata (i.e., information about the table data) are stored in the header. For example, the keyword TFORM1 contains the format of the first field, TTYPE2 the name of the second field, etc. NAXIS2 gives the number of records (rows) and TFIELDS gives the number of fields (columns). For FITS tables, the maximum number of fields is 999. The data type specified in TFORM is represented by letter codes for binary tables and a Fortran-like format string for ASCII tables. Note that this is different from the format specifications when constructing a record array.

Reading a FITS Table

Like images, the `.data` attribute of a table HDU contains the data of the table.

Example

To read a FITS Table:

```
>>> from astropy.io import fits
>>> fits_table_filename =
fits.util.get_testdata_filepath('btable.fits')

>>> hdul = fits.open(fits_table_filename)  # open a FITS file
>>> data = hdul[1].data  # assume the first extension is a table
>>> # show the first two rows
>>> first_two_rows = data[:2]
>>> first_two_rows
[(1, 'Sirius', -1.45000005, 'A1V') (2, 'Canopus', -0.73000002,
'F0Ib')]
>>> # show the values in field "mag"
>>> magnitudes = data['mag']
>>> magnitudes
array([-1.45000005, -0.73000002, -0.1       ], dtype=float32)
>>> # columns can be referenced by index too
>>> names = data.field(1)
>>> names.tolist()
['Sirius', 'Canopus', 'Rigil Kent']
>>> hdul.close()
```

Note that in `astropy`, when using the `field()` method, it is 0-indexed while the suffixes in header keywords such as TFORM is 1-indexed. So, `data.field(0)` is the data in the column with the name specified in TTYPE1 and format in TFORM1.

Warning

The FITS format allows table columns with a zero-width data format, such as `'0D'`. This is probably intended as a space-saving measure on files in which that column contains no data. In such files, the zero-width columns are omitted when accessing the table data, so the indexes of fields might change when using the `field()` method. For this reason, if you expect to encounter files containing zero-width columns it is recommended to access fields by name rather than by index.

Table Operations

Selecting Records in a Table

Like image data, we can use the same “mask array” idea to pick out desired records from a table and make a new table out of it.

Examples

Assuming the table’s second field as having the name ‘magnitude’, an output table containing all the records of magnitude > -0.5 from the input table is generated:

```
>>> with fits.open(fits_table_filename) as hdul:
...     data = hdul[1].data
...     mask = data['mag'] > -0.5
...     newdata = data[mask]
...     hdu = fits.BinTableHDU(data=newdata)
...     hdu.writeto('newtable.fits')
```

It is also possible to update the data from the HDU object in-place:

```
>>> with fits.open(fits_table_filename) as hdul:
...     hdu = hdul[1]
...     mask = hdu.data['mag'] > -0.5
...     hdu.data = hdu.data[mask]
...     hdu.writeto('newtable2.fits')
```

Merging Tables

Merging different tables is very convenient in `astropy`.

Examples

To merge the column definitions of the input tables:

```
>>> fits_other_table_filename =
fits.util.get_testdata_filepath('table.fits')

>>> with fits.open(fits_table_filename) as hdul1:
...     with fits.open(fits_other_table_filename) as hdul2:
...         new_columns = hdul1[1].columns + hdul2[1].columns
...         new_hdu = fits.BinTableHDU.from_columns(new_columns)
>>> new_columns
ColDefs(
        name = 'order'; format = 'I'
        name = 'name'; format = '20A'
        name = 'mag'; format = 'E'
        name = 'Sp'; format = '10A'
        name = 'target'; format = '20A'
        name = 'V_mag'; format = 'E'
    )
```

The number of fields in the output table will be the sum of numbers of fields of the input tables. Users have to make sure the input tables do not share any common field names. The number of records in the output table will be the largest number of records of all input tables. The expanded slots for the originally shorter table(s) will be zero (or blank) filled.

Another version of this example can be used to append a new column to a table. Updating an existing table with a new column is generally more difficult than it is worth, but you can “append” a column to a table by creating a new table with columns from the existing table plus the new column(s):

```
>>> with fits.open(fits_table_filename) as hdul:
...     orig_table = hdul[1].data
...     orig_cols = orig_table.columns
>>> new_cols = fits.ColDefs([
...     fits.Column(name='NEWCOL1', format='D',
...                 array=np.zeros(len(orig_table))),
...     fits.Column(name='NEWCOL2', format='D',
...                 array=np.zeros(len(orig_table)))])
>>> hdu = fits.BinTableHDU.from_columns(orig_cols + new_cols)
```

Now `newtable.fits` contains a new table with the original table, plus the two new columns filled with zeros.

Appending Tables

Appending one table after another is slightly trickier, since the two tables may have different field attributes.

Examples

Here, the first example is to append by field indices, and the second one is to append by field names. In both cases, the output table will inherit the column attributes (name, format, etc.) of the first table:

```
>>> with fits.open(fits_table_filename) as hdul1:
...     with fits.open(fits_table_filename) as hdul2:
...         nrows1 = hdul1[1].data.shape[0]
...         nrows2 = hdul2[1].data.shape[0]
...         nrows = nrows1 + nrows2
...         hdu = fits.BinTableHDU.from_columns(hdul1[1].columns,
nrows=nrows)
...         for colname in hdul1[1].columns.names:
...             hdu.data[colname][nrows1:] = hdul2[1].data[colname]
```

Scaled Data in Tables

A table field’s data, like an image, can also be scaled. Scaling in a table has a more generalized meaning than in images. In images, the physical data is a simple linear transformation from the storage data. The table fields do have such a construct too, where BSCALE and BZERO are stored in the header as TSCALn and TZEROn. In addition, boolean columns and ASCII tables’ numeric fields are also generalized “scaled” fields, but without TSCAL and TZERO.

All scaled fields, like the image case, will take extra memory space as well as processing. So, if high performance is desired, try to minimize the use of scaled fields.

All of the scalings are done for the user, so the user only sees the physical data. Thus, there is no need to worry about scaling back and forth between the physical and storage column values.

Creating a FITS Table

Column Creation

To create a table from scratch, it is necessary to create individual columns first. A `Column` constructor needs the minimal information of column name and format. Here is a summary of all allowed formats for a binary table:

FITS format code	Description	8-bit bytes
L	logical (Boolean)	1
X	bit	*
B	Unsigned byte	1
I	16-bit integer	2
J	32-bit integer	4
K	64-bit integer	8
A	character	1
E	single precision floating point	4
D	double precision floating point	8
C	single precision complex	8
M	double precision complex	16
P	array descriptor	8
Q	array descriptor	16

We will concentrate on binary tables in this chapter. ASCII tables will be discussed in a later chapter. The less frequently used X format (bit array) and P format (used in variable length tables) will also be discussed in a later chapter.

Besides the required name and format arguments in constructing a `Column`, there are many optional arguments which can be used in creating a column. Here is a list of these arguments and their corresponding header keywords and descriptions:

Argument	Corresponding	Description

```
in Column()     header keyword

name            TTYPE                     column name
format          TFORM                     column format
unit            TUNIT                     unit
null            TNULL                     null value (only for B, I, and
J)
bscale          TSCAL                     scaling factor for data
bzero           TZERO                     zero point for data scaling
disp            TDISP                     display format
dim             TDIM                      multi-dimensional array spec
start           TBCOL                     starting position for ASCII
table
coord_type      TCTYP                     coordinate/axis type
coord_unit      TCUNI                     coordinate/axis unit
coord_ref_point TCRPX                     pixel coordinate of the
reference point
coord_ref_value TCRVL                     coordinate value at reference
point
coord_inc       TCDLT                     coordinate increment at
reference point
time_ref_pos    TRPOS                     reference position for a time
coordinate column
ascii                                     specifies a column for an ASCII
table
array                                     the data of the column
```

Examples

Here are a few Columns using various combinations of the optional arguments:

```
>>> counts = np.array([312, 334, 308, 317])
>>> names = np.array(['NGC1', 'NGC2', 'NGC3', 'NGC4'])
>>> values = np.arange(2*2*4).reshape(4, 2, 2)
>>> col1 = fits.Column(name='target', format='10A', array=names)
>>> col2 = fits.Column(name='counts', format='J', unit='DN',
array=counts)
>>> col3 = fits.Column(name='notes', format='A10')
>>> col4 = fits.Column(name='spectrum', format='10E')
>>> col5 = fits.Column(name='flag', format='L', array=[True, False,
True, True])
>>> col6 = fits.Column(name='intarray', format='4I', dim='(2, 2)',
array=values)
```

In this example, formats are specified with the FITS letter codes. When there is a number (>1) preceding a (numeric type) letter code, it means each cell in that field is a one-dimensional array. In the case of column “col4”, each cell is an array (a NumPy array) of 10 elements. And in the case of column “col6”, with the use of the “dim” argument, each cell is a multi-dimensional array of 2x2 elements.

For character string fields, the number should be to the *left* of the letter ‘A’ when creating binary tables, and should be to the *right* when creating ASCII tables. However, as this is a common confusion, both formats are understood when creating binary tables (note, however, that upon writing to a file the correct format will be written in the header). So, for columns “col1” and “col3”, they both have 10 characters in each of their cells. For numeric data type, the dimension number must be before the letter code, not after.

After the columns are constructed, the `BinTableHDU.from_columns()` class method can be used to construct a table HDU. We can either go through the column definition object:

```
>>> coldefs = fits.ColDefs([col1, col2, col3, col4, col5, col6])
>>> hdu = fits.BinTableHDU.from_columns(coldefs)
>>> coldefs
ColDefs(
    name = 'target'; format = '10A'
    name = 'counts'; format = 'J'; unit = 'DN'
    name = 'notes'; format = '10A'
    name = 'spectrum'; format = '10E'
    name = 'flag'; format = 'L'
    name = 'intarray'; format = '4I'; dim = '(2, 2)'
)
```

or directly use the `BinTableHDU.from_columns()` method:

```
>>> hdu = fits.BinTableHDU.from_columns([col1, col2, col3, col4,
col5, col6])
>>> hdu.columns
ColDefs(
    name = 'target'; format = '10A'
    name = 'counts'; format = 'J'; unit = 'DN'
    name = 'notes'; format = '10A'
    name = 'spectrum'; format = '10E'
    name = 'flag'; format = 'L'
    name = 'intarray'; format = '4I'; dim = '(2, 2)'
)
```

Note

Users familiar with older versions of `astropy` will wonder what happened to `astropy.io.fits.new_table`. `BinTableHDU.from_columns()` and its companion for ASCII tables

TableHDU.from_columns() are the same in the arguments they accept and their behavior, but make it more explicit as to what type of table HDU they create.

A look at the newly created HDU's header will show that relevant keywords are properly populated:

```
>>> hdu.header
XTENSION= 'BINTABLE'           / binary table extension
BITPIX  =                    8 / array data type
NAXIS   =                    2 / number of array dimensions
NAXIS1  =                   73 / length of dimension 1
NAXIS2  =                    4 / length of dimension 2
PCOUNT  =                    0 / number of group parameters
GCOUNT  =                    1 / number of groups
TFIELDS =                    6 / number of table fields
TTYPE1  = 'target  '
TFORM1  = '10A     '
TTYPE2  = 'counts  '
TFORM2  = 'J       '
TUNIT2  = 'DN      '
TTYPE3  = 'notes   '
TFORM3  = '10A     '
TTYPE4  = 'spectrum'
TFORM4  = '10E     '
TTYPE5  = 'flag    '
TFORM5  = 'L       '
TTYPE6  = 'intarray'
TFORM6  = '4I      '
TDIM6   = '(2, 2)  '
```

Warning

It should be noted that when creating a new table with BinTableHDU.from_columns(), an in-memory copy of all of the input column arrays is created. This is because it is not guaranteed that the columns are arranged contiguously in memory in row-major order (in fact, they are most likely not), so they have to be combined into a new array.

However, if the array data *is* already contiguous in memory, such as in an existing record array, a kludge can be used to create a new table HDU without any copying. First, create the Columns as before, but without using the `array=` argument:

```
>>> col1 = fits.Column(name='target', format='10A')
```

Then call BinTableHDU.from_columns():

```
>>> hdu = fits.BinTableHDU.from_columns([col1, col2, col3, col4,
col5])
```

This will create a new table HDU as before, with the correct column definitions, but an empty data section. Now you can assign your array directly to the HDU's data attribute:

```
>>> hdu.data = mydata
```

In a future version of `astropy`, table creation will be simplified and this process will not be necessary.

FITS Tables with Time Columns

The FITS Time standard paper defines the formats and keywords used to represent timing information in FITS files. The `astropy` FITS package provides support for reading and writing native **`Time`** columns and objects using this format. This is done within the FITS unified I/O interface and examples of usage can be found in the TDISPn Keyword section. The support is not complete and only a subset of the full standard is implemented.

Example

The following is an example of a Header extract of a binary table (event list) with a time column:

```
COMMENT      ---------- Globally valid key words ----------------
TIMESYS = 'TT      '          / Time system
MJDREF  = 50814.000000000000  / MJD zero point for (native) TT (=
1998-01-01)
MJD-OBS = 53516.257939301       / MJD for observation in (native) TT

COMMENT      ---------- Time Column -----------------------
TTYPE1  = 'Time    '          / S/C TT corresponding to mid-exposure
TFORM1  = '2D      '          / format of field
TUNIT1  = 's       '
TCTYP1  = 'TT      '
TCNAM1  = 'Terrestrial Time'  / This is TT
TCUNI1  = 's       '
```

However, the FITS standard and the `astropy` Time object are not perfectly mapped and some compromises must be made. To help the user understand how the `astropy` code deals with these situations, the following text describes the approach that `astropy` takes in some detail.

To create FITS columns which adhere to the FITS Time standard, we have taken into account the following important points stated in the FITS Time paper.

The strategy used to store **`Time`** columns in FITS tables is to create a **`Header`** with the appropriate time coordinate global reference keywords and the column-specific override keywords. The module `astropy.io.fits.fitstime` deals with the reading and writing of Time columns.

The following keywords set the Time Coordinate Frame:

- TIME SCALE

  The most important of all of the metadata is the time scale which is a specification for measuring time.

  ```
  TIMESYS (string-valued)
  Time scale; default UTC

  TCTYPn (string-valued)
  Column-specific override keyword
  ```

  The global time scale may be overridden by a time scale recorded in the table equivalent keyword `TCTYPn` for time coordinates in FITS table columns. `TCTYna` is used for alternate coordinates.

- TIME REFERENCE

  The reference point in time to which all times in the HDU are relative. Since there are no context-specific reference times in case there are multiple time columns in the same table, we need to adjust the reference times for the columns using some other keywords.

  The reference point in time shall be specified through one of the three following keywords, which are listed in decreasing order of preference:

  ```
  MJDREF (floating-valued)
  Reference time in MJD

  JDREF (floating-valued)
  Reference time in JD

  DATEREF (datetime-valued)
  Reference time in ISO-8601
  ```

  The time reference keywords (MJDREF, JDREF, DATEREF) are interpreted using the time scale specified in `TIMESYS`.

  Note

If none of the three keywords are present, there is no problem as long as all times in the HDU are expressed in ISO-8601 `Datetime Strings` format: `CCYY-MM-DD[Thh:mm:ss[.s...]]` (e.g., `"2015-04-05T12:22:33.8"`); otherwise MJDREF = 0.0 must be assumed.

The value of the reference time has global validity for all time values, but it does not have a particular time scale associated with it. Thus we need to use `TCRVLn` (time coordinate reference value) keyword to compensate for the time scale differences.

- TIME REFERENCE POSITION

  The reference position, specified by the keyword `TREFPOS`, specifies the spatial location at which the time is valid, either where the observation was made or the point in space for which light-time corrections have been applied. This may be a standard location (such as `GEOCENTER` or `TOPOCENTER`) or a point in space defined by specific coordinates.

```
TREFPOS (string-valued)
Time reference position; default TOPOCENTER

TRPOSn (string-valued)
Column-specific override keyword
```

Note

For TOPOCENTER, we need to specify the observatory location (ITRS Cartesian coordinates or geodetic latitude/longitude/height) in the `OBSGEO-*` keywords.

- TIME REFERENCE DIRECTION

  If any pathlength corrections have been applied to the time stamps (i.e., if the reference position is not `TOPOCENTER` for observational data), the reference direction that is used in calculating the pathlength delay should be provided in order to maintain a proper analysis trail of the data. However, this is useful only if there is also information available on the location from where the observation was made (the observatory location).

  The reference direction is indicated through a reference to specific keywords. These keywords may explicitly hold the direction or indicate columns holding the coordinates.

```
TREFDIR (string-valued)
Pointer to time reference direction

TRDIRn (string-valued)
```

```
Column-specific override keyword
```

- TIME UNIT

  The FITS standard recommends the time unit to be one of the allowed ones in the specification.

  ```
  TIMEUNIT (string-valued)
  Time unit; default s

  TCUNIn (string-valued)
  Column-specific override
  ```

- TIME OFFSET

  It is sometimes convenient to be able to apply a uniform clock correction in bulk by putting that number in a single keyword. A second use for a time offset is to set a zero offset to a relative time series, allowing zero-relative times, or higher precision, in the time stamps. Its default value is zero.

  ```
  TIMEOFFS (floating-valued)
  This has global validity
  ```

- The absolute, relative errors and time resolution, time binning can be used when needed.

The following keywords define the global time informational keywords:

- DATE and DATE-* keywords

  These define the date of HDU creation and observation in ISO-8601. `DATE` is in UTC if the file is constructed on the Earth's surface and others are in the time scale given by `TIMESYS`.

- MJD-* keywords

  These define the same as above, but in `MJD` (Modified Julian Date).

The implementation writes a subset of the above FITS keywords, which map to the Time metadata. Time is intrinsically a coordinate and hence shares keywords with the `World Coordinate System` specification for spatial coordinates. Therefore, while reading FITS tables with time columns, the verification that a coordinate column is indeed time is done using the FITS WCS standard rules and suggestions.

Verification

`astropy` has built in a flexible scheme to verify FITS data conforming to the FITS standard. The basic verification philosophy in `astropy` is to be tolerant with input and strict with output.

When `astropy` reads a FITS file which does not conform to FITS standard, it will not raise an error and exit. It will try to make the best educated interpretation and only gives up when the offending data is accessed and no unambiguous interpretation can be reached.

On the other hand, when writing to an output FITS file, the content to be written must be strictly compliant to the FITS standard by default. This default behavior can be overwritten by several other options, so the user will not be held up because of a minor standard violation.

FITS Standard

Since FITS standard is a “loose” standard, there are many places the violation can occur and to enforce them all will be almost impossible. It is not uncommon for major observatories to generate data products which are not 100% FITS compliant. Some observatories have also developed their own nonstandard dialect and some of these are so prevalent that they have become de facto standards. Examples include the long string value and the use of the CONTINUE card.

The violation of the standard can happen at different levels of the data structure. `astropy`’s verification scheme is developed on these hierarchical levels. Here are the three `astropy` verification levels:

1. The HDU List
2. Each HDU
3. Each Card in the HDU Header

These three levels correspond to the three categories of objects: **`HDUList`**, any HDU (e.g., **`PrimaryHDU`**, **`ImageHDU`**, etc.), and **`Card`**. They are the only objects having the `verify()` method. Most other classes in **`astropy.io.fits`** do not have a `verify()` method.

If `verify()` is called at the HDU List level, it verifies standard compliance at all three levels, but a call of `verify()` at the Card level will only check the compliance of that Card. Since `astropy` is tolerant when reading a FITS file, no `verify()` is called on input. On output, `verify()` is called with the most restrictive option as the default.

Verification Options

There are several options accepted by all verify(option) calls in `astropy`. In addition, they available for the `output_verify` argument of the following methods: `close()`, `writeto()`, and `flush()`. In these cases, they are passed to a `verify()` call within these methods. The available options are:

exception

This option will raise an exception if any FITS standard is violated. This is the default option for output (i.e., when `writeto()`, `close()`, or `flush()` is called). If a user wants to overwrite this default on output, the other options listed below can be used.

warn

This option is the same as the ignore option but will send warning messages. It will not try to fix any FITS standard violations whether fixable or not.

ignore

This option will ignore any FITS standard violation. On output, it will write the HDU List content to the output FITS file, whether or not it is conforming to the FITS standard.

The ignore option is useful in the following situations:

1. An input FITS file with nonstandard formatting is read and the user wants to copy or write out to an output file. The nonstandard formatting will be preserved in the output file.
2. A user wants to create a nonstandard FITS file on purpose, possibly for testing or consistency.

No warning message will be printed out. This is like a silent warning option (see below).

fix

This option will try to fix any FITS standard violations. It is not always possible to fix such violations. In general, there are two kinds of FITS standard violations: fixable and non-fixable. For example, if a keyword has a floating number with an exponential notation in lower case ‘e’ (e.g., 1.23e11) instead of the upper case ‘E’ as required by the FITS standard, it is a fixable violation. On the other hand, a keyword name like ‘P.I.’ is not fixable, since it will not know what to use to replace the disallowed periods. If a violation is fixable, this option will print out a message noting it is fixed. If it is not fixable, it will throw an exception.

The principle behind fixing is to do no harm. For example, it is plausible to ‘fix’ a Card with a keyword name like ‘P.I.’ by deleting it, but `astropy` will not take such action to hurt the integrity of the data.

Not all fixes may be the “correct” fix, but at least `astropy` will try to make the fix in such a way that it will not throw off other FITS readers.

silentfix

Same as fix, but will not print out informative messages. This may be useful in a large script where the user does not want excessive harmless messages. If the violation is not fixable, it will still throw an exception.

In addition, as of `astropy` version 0.4.0 the following combined options are available:

- **fix+ignore**
- **fix+warn**
- **fix+exception**
- **silentfix+ignore**
- **silentfix+warn**
- **silentfix+exception**

These options combine the semantics of the basic options. For example, `silentfix+exception` is actually equivalent to just `silentfix` in that fixable errors will be fixed silently, but any unfixable errors will raise an exception. On the other hand, `silentfix+warn` will issue warnings for unfixable errors, but will stay silent about any fixed errors.

Verifications at Different Data Object Levels

We will examine what `astropy`’s verification does at the three different levels:

Verification at HDUList

At the HDU List level, the verification is only for two simple cases:

1. Verify that the first HDU in the HDU list is a primary HDU. This is a fixable case. The fix is to insert a minimal primary HDU into the HDU list.
2. Verify the second or later HDU in the HDU list is not a primary HDU. Violation will not be fixable.

Verification at Each HDU

For each HDU, the mandatory keywords, their locations in the header, and their values will be verified. Each FITS HDU has a fixed set of required keywords in a fixed order. For example, the primary HDU’s header must at least have the following keywords:

```
SIMPLE =                    T /
```

```
BITPIX =                    8 /
NAXIS  =                    0
```

If any of the mandatory keywords are missing or in the wrong order, the fix option will fix them:

```
>>> hdu.header                 # has a 'bad' header
SIMPLE =                    T /
NAXIS  =                    0
BITPIX =                    8 /
>>> hdu.verify('fix')          # fix it
Output verification result:
'BITPIX' card at the wrong place (card 2). Fixed by moving it to the
right
place (card 1).
>>> hdu.header                   # voila!
SIMPLE =                    T / conforms to FITS standard
BITPIX =                    8 / array data type
NAXIS  =                    0
```

Verification at Each Card

The lowest level, the Card, also has the most complicated verification possibilities.

Examples

Here is a list of fixable and not fixable Cards:

Fixable Cards:

1. Floating point numbers with lower case ‘e’ or ‘d’:

```
>>> from astropy.io import fits
>>> c = fits.Card.fromstring('FIX1    = 2.1e23')
>>> c.verify('silentfix')
>>> print(c)
FIX1    =              2.1E23
```

2. The equal sign is before column nine in the card image:

```
>>> c = fits.Card.fromstring('FIX2= 2')
>>> c.verify('silentfix')
>>> print(c)
FIX2    =                   2
```

3. String value without enclosing quotes:

```
>>> c = fits.Card.fromstring('FIX3    = string value without
quotes')
>>> c.verify('silentfix')
>>> print(c)
FIX3    = 'string value without quotes'
```

4. Missing equal sign before column nine in the card image.
5. Space between numbers and E or D in floating point values:

```
>>> c = fits.Card.fromstring('FIX5    = 2.4 e 03')
>>> c.verify('silentfix')
>>> print(c)
FIX5    =               2.4E03
```

6. Unparsable values will be “fixed” as a string:

```
>>> c = fits.Card.fromstring('FIX6    = 2 10 ')
>>> c.verify('fix+warn')
>>> print(c)
FIX6    = '2 10    '
```

Unfixable Cards:

1. Illegal characters in keyword name.

We will summarize the verification with a “life-cycle” example:

```
>>> h = fits.PrimaryHDU()  # create a PrimaryHDU
>>> # Try to add an non-standard FITS keyword 'P.I.' (FITS does no
allow
>>> # '.' in the keyword), if using the update() method - doesn't
work!
>>> h['P.I.'] = 'Hubble'
ValueError: Illegal keyword name 'P.I.'
>>> # Have to do it the hard way (so a user will not do this by
accident)
>>> # First, create a card image and give verbatim card content
(including
>>> # the proper spacing, but no need to add the trailing blanks)
>>> c = fits.Card.fromstring("P.I. = 'Hubble'")
>>> h.header.append(c)  # then append it to the header
>>> # Now if we try to write to a FITS file, the default output
>>> # verification will not take it.
>>> h.writeto('pi.fits')
Output verification result:
HDU 0:
  Card 4:
    Unfixable error: Illegal keyword name 'P.I.'
```

```
......
  raise VerifyError
VerifyError
>>> # Must set the output_verify argument to 'ignore', to force
writing a
>>> # non-standard FITS file
>>> h.writeto('pi.fits', output_verify='ignore')
>>> # Now reading a non-standard FITS file
>>> # astropy.io.fits is magnanimous in reading non-standard FITS
files
>>> hdus = fits.open('pi.fits')
>>> hdus[0].header
SIMPLE =                    T / conforms to FITS standard
BITPIX =                    8 / array data type
NAXIS  =                    0 / number of array dimensions
EXTEND =                    T
P.I.   = 'Hubble'
>>> # even when you try to access the offending keyword, it does NOT
>>> # complain
>>> hdus[0].header['p.i.']
'Hubble'
>>> # But if you want to make sure if there is anything wrong/non-
standard,
>>> # use the verify() method
>>> hdus.verify()
Output verification result:
HDU 0:
  Card 4:
    Unfixable error: Illegal keyword name 'P.I.'
```

Verification Using the FITS Checksum Keyword Convention

The North American FITS committee has reviewed the FITS Checksum Keyword Convention for possible adoption as a FITS Standard. This convention provides an integrity check on information contained in FITS HDUs. The convention consists of two header keyword cards: CHECKSUM and DATASUM. The CHECKSUM keyword is defined as an ASCII character string whose value forces the 32-bit 1’s complement checksum accumulated over all the 2880-byte FITS logical records in the HDU to equal negative zero. The DATASUM keyword is defined as a character string containing the unsigned integer value of the 32-bit 1’s complement checksum of the data records in the HDU. Verifying the accumulated checksum is still equal to negative zero provides a fairly reliable way to determine that the HDU has not been modified by subsequent data processing operations or corrupted while copying or storing the file on physical media.

In order to avoid any impact on performance, by default `astropy` will not verify HDU checksums when a file is opened or generate checksum values when a file is written. In fact, CHECKSUM and DATASUM cards are automatically removed from HDU headers when a file is opened, and any CHECKSUM or DATASUM cards are stripped from headers when an HDU is written to a file. In order to verify the checksum values for HDUs when opening a file, the user must supply the checksum keyword argument in the call to the open convenience function with a value of True. When this is done, any checksum verification failure will cause a warning to be issued (via the warnings module). If checksum verification is requested in the open, and no CHECKSUM or DATASUM cards exist in the HDU header, the file will open without comment. Similarly, in order to output the CHECKSUM and DATASUM cards in an HDU header when writing to a file, the user must supply the checksum keyword argument with a value of True in the call to the `writeto()` function. It is possible to write only the DATASUM card to the header by supplying the checksum keyword argument with a value of 'datasum'.

Examples

To verify the checksum values for HDUs when opening a file:

```
>>> # Open the file pix.fits verifying the checksum values for all
HDUs
>>> hdul = fits.open('pix.fits', checksum=True)
```

```
>>> # Open the file in.fits where checksum verification fails for the
>>> # primary HDU
>>> hdul = fits.open('in.fits', checksum=True)
Warning:  Checksum verification failed for HDU #0.
```

```
>>> # Create file out.fits containing an HDU constructed from data
and
>>> # header containing both CHECKSUM and DATASUM cards.
>>> fits.writeto('out.fits', data, header, checksum=True)
```

```
>>> # Create file out.fits containing all the HDUs in the HDULIST
>>> # hdul with each HDU header containing only the DATASUM card
>>> hdul.writeto('out.fits', checksum='datasum')
```

```
>>> # Create file out.fits containing the HDU hdu with both CHECKSUM
>>> # and DATASUM cards in the header
>>> hdu.writeto('out.fits', checksum=True)
>>> # Append a new HDU constructed from array data to the end of
>>> # the file existingfile.fits with only the appended HDU
>>> # containing both CHECKSUM and DATASUM cards.
>>> fits.append('existingfile.fits', data, checksum=True)
```

Less Familiar Objects

In this chapter, we will discuss less frequently used FITS data structures. They include ASCII tables, variable length tables, and random access group FITS files.

ASCII Tables

FITS standard supports both binary and ASCII tables. In ASCII tables, all of the data are stored in a human-readable text form, so it takes up more space and extra processing to parse the text for numeric data. Depending on how the columns are formatted, floating point data may also lose precision.

In `astropy`, the interface for ASCII tables and binary tables is basically the same (i.e., the data is in the `.data` attribute and the `field()` method is used to refer to the columns and returns a `numpy` array). When reading the table, `astropy` will automatically detect what kind of table it is.

```
>>> from astropy.io import fits
>>> filename = fits.util.get_testdata_filepath('ascii.fits')
>>> hdul = fits.open(filename)
>>> hdul[1].data[:1]
FITS_rec([(10.123, 37)],
         dtype=(numpy.record, {'names':['a','b'], 'formats':
['S10','S5'], 'offsets':[0,11], 'itemsize':16}))
>>> hdul[1].data['a']
array([  10.123,    5.2  ,   15.61 ,    0.    ,  345.   ])
>>> hdul[1].data.formats
['E10.4', 'I5']
>>> hdul.close()
```

Note that the formats in the record array refer to the raw data which are ASCII strings (therefore ‘a11’ and ‘a5’), but the `.formats` attribute of data retains the original format specifications (‘E10.4’ and ‘I5’).

Creating an ASCII Table

Creating an ASCII table from scratch is similar to creating a binary table. The difference is in the Column definitions. The columns/fields in an ASCII table are more limited than in a binary table. It does not allow more than one numerical value in a cell. Also, it only supports a subset of what is allowed in a binary table, namely character strings, integer, and (single and double precision) floating point numbers. Boolean and complex numbers are not allowed.

The format syntax (the values of the TFORM keywords) is different from that of a binary table. They are:

```
Aw          Character string
Iw          (Decimal) Integer
Fw.d        Double precision real
Ew.d        Double precision real, in exponential notation
Dw.d        Double precision real, in exponential notation
```

where w is the width, and d the number of digits after the decimal point. The syntax difference between ASCII and binary tables can be confusing. For example, a field of 3-character string is specified as '3A' in a binary table and as 'A3' in an ASCII table.

The other difference is the need to specify the table type when using the `TableHDU.from_columns()` method, and that `Column` should be provided the `ascii=True` argument in order to be unambiguous.

Note

Although binary tables are more common in most FITS files, earlier versions of the FITS format only supported ASCII tables. That is why the class `TableHDU` is used for representing ASCII tables specifically, whereas `BinTableHDU` is more explicit that it represents a binary table. These names come from the value `XTENSION` keyword in the tables' headers, which is `TABLE` for ASCII tables and `BINTABLE` for binary tables.

`TableHDU.from_columns()` can be used like so:

```
>>> import numpy as np

>>> a1 = np.array(['abcd', 'def'])
>>> r1 = np.array([11., 12.])
>>> col1 = fits.Column(name='abc', format='A3', array=a1, ascii=True)
>>> col2 = fits.Column(name='def', format='E', array=r1, bscale=2.3,
...                    bzero=0.6, ascii=True)
>>> col3 = fits.Column(name='t1', format='I', array=[91, 92, 93],
ascii=True)
>>> hdu = fits.TableHDU.from_columns([col1, col2, col3])
>>> hdu.data
FITS_rec([('abc', 11.0, 91), ('def', 12.0, 92), ('', 0.0, 93)],
         dtype=(numpy.record, [('abc', 'S3'), ('def', 'S15'), ('t1',
'S10')]))
```

It should be noted that when the formats of the columns are unambiguously specific to ASCII tables it is not necessary to specify `ascii=True` in the **`ColDefs`** constructor. In this case there *is* ambiguity because the format code `'I'` represents a 16-bit integer in binary tables, while in ASCII tables it is not technically a valid format. ASCII table format codes technically require a character width for each column, such as `'I10'` to create a column that can hold integers up to 10 characters wide.

However, `astropy` allows the width specification to be omitted in some cases. When it is omitted from `'I'` format columns the minimum width needed to accurately represent all integers in the column is used. The only problem with using this shortcut is its ambiguity with the binary table `'I'` format, so specifying `ascii=True` is a good practice (though `astropy` will still figure out what you meant in most cases).

Variable Length Array Tables

The FITS standard also supports variable length array tables. The basic idea is that sometimes it is desirable to have tables with cells in the same field (column) that have the same data type but have different lengths/dimensions. Compared with the standard table data structure, the variable length table can save storage space if there is a large dynamic range of data lengths in different cells.

A variable length array table can have one or more fields (columns) which are variable length. The rest of the fields (columns) in the same table can still be regular, fixed-length ones. `astropy` will automatically detect what kind of field it is during reading; no special action is needed from the user. The data type specification (i.e., the value of the TFORM keyword) uses an extra letter ‘P’ and the format is:

```
rPt(max)
```

where `r` may be 0 or 1 (typically omitted, as it is not applicable to variable length arrays), `t` is one of the letter codes for basic data types (L, B, I, J, etc.; currently, the X format is not supported for variable length array field in `astropy`), and `max` is the maximum number of elements of any array in the column. So, for a variable length field of int16, the corresponding format spec is, for example, ‘PJ(100)’.

Example

This example shows a variable length array field of data type int16:

```
>>> filename =
fits.util.get_testdata_filepath('variable_length_table.fits')
>>> hdul = fits.open(filename)
>>> hdul[1].header['tform1']
'PI(3)'
>>> print(hdul[1].data.field(0))
[array([45, 56], dtype=int16) array([11, 12, 13], dtype=int16)]
>>> hdul.close()
```

In this field the first row has one element, the second row has two elements, etc. Accessing variable length fields is almost identical to regular fields, except that operations on the whole field simultaneously are usually not possible. A user has to process the field row by row as though they are independent arrays.

Creating a Variable Length Array Table

Creating a variable length table is almost identical to creating a regular table. The only difference is in the creation of field definitions which are variable length arrays. First, the data type specification will need the ‘P’ letter, and secondly, the field data must be an objects array (as included in the `numpy` module).

Example

Here is an example of creating a table with two fields; one is regular and the other a variable length array:

```
>>> col1 = fits.Column(
...    name='var', format='PI()',
...    array=np.array([[45, 56], [11, 12, 13]], dtype=np.object_))
>>> col2 = fits.Column(name='xyz', format='2I', array=[[11, 3], [12,
4]])
>>> hdu = fits.BinTableHDU.from_columns([col1, col2])
>>> data = hdu.data
>>> data
FITS_rec([([45, 56], [11,  3]), ([11, 12, 13], [12,  4])],
         dtype=(numpy.record, [('var', '<i4', (2,)), ('xyz', '<i2',
(2,))]))
>>> hdu.writeto('variable_length_table.fits')
>>> with fits.open('variable_length_table.fits') as hdul:
...     print(repr(hdul[1].header))
XTENSION= 'BINTABLE'           / binary table extension
BITPIX  =                    8 / array data type
NAXIS   =                    2 / number of array dimensions
NAXIS1  =                   12 / length of dimension 1
NAXIS2  =                    2 / length of dimension 2
```

```
PCOUNT  =                   10 / number of group parameters
GCOUNT  =                    1 / number of groups
TFIELDS =                    2 / number of table fields
TTYPE1  = 'var     '
TFORM1  = 'PI(3)   '
TTYPE2  = 'xyz     '
TFORM2  = '2I      '
```

Random Access Groups

Another less familiar data structure supported by the FITS standard is the random access group. This convention was established before the binary table extension was introduced. In most cases its use can now be superseded by the binary table. It is mostly used in radio interferometry.

Like primary HDUs, a Random Access Group HDU is always the first HDU of a FITS file. Its data has one or more groups. Each group may have any number (including 0) of parameters, together with an image. The parameters and the image have the same data type.

All groups in the same HDU have the same data structure, that is, same data type (specified by the keyword BITPIX, as in image HDU), same number of parameters (specified by PCOUNT), and the same size and shape (specified by NAXISn keywords) of the image data. The number of groups is specified by GCOUNT and the keyword NAXIS1 is always 0. Thus the total data size for a Random Access Group HDU is:

```
|BITPIX| * GCOUNT * (PCOUNT + NAXIS2 * NAXIS3 * ... * NAXISn)
```

Header and Summary
Accessing the header of a Random Access Group HDU is no different from any other HDU; you can use the .header attribute.

The content of the HDU can similarly be summarized by using the **HDUList.info()** method:

```
>>> filename = fits.util.get_testdata_filepath('group.fits')
>>> hdul = fits.open(filename)
>>> hdul[0].header['groups']
True
>>> hdul[0].header['gcount']
10
>>> hdul[0].header['pcount']
3
>>> hdul.info()
```

```
Filename: ...group.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 GroupsHDU      15   (5, 3, 1, 1)   float32   10
Groups  3 Parameters
```

Data: Group Parameters

The data part of a Random Access Group HDU is, like other HDUs, in the `.data` attribute. It includes both parameter(s) and image array(s).

Examples

To show the contents of the third group, including parameters and data:

```
>>> hdul[0].data[2]
(2.0999999, 42.0, 42.0, array([[[[30., 31., 32., 33., 34.],
         [35., 36., 37., 38., 39.],
         [40., 41., 42., 43., 44.]]]], dtype=float32))
```

The data first lists all of the parameters, then the image array, for the specified group(s). As a reminder, the image data in this file has the shape of (1,1,1,4,3) in Python or C convention, or (3,4,1,1,1) in IRAF or Fortran convention.

To access the parameters, first find out what the parameter names are, with the `.parnames` attribute:

```
>>> hdul[0].data.parnames # get the parameter names
['abc', 'xyz', 'xyz']
```

The group parameter can be accessed by the `par()` method. Like the table `field()` method, the argument can be either index or name:

```
>>> hdul[0].data.par(0)[8]  # Access group parameter by name or by
index
8.1
>>> hdul[0].data.par('abc')[8]
8.1
```

Note that the parameter name 'xyz' appears twice. This is a feature in the random access group, and it means to add the values together. Thus:

```
>>> hdul[0].data.parnames  # get the parameter names
['abc', 'xyz', 'xyz']
>>> hdul[0].data.par(1)[8]  # Duplicate parameter name 'xyz'
42.0
>>> hdul[0].data.par(2)[8]
42.0
>>> # When accessed by name, it adds the values together if the name
```

```
is
>>> # shared by more than one parameter
>>> hdul[0].data.par('xyz')[8]
84.0
```

The `par()` is a method for either the entire data object or one data item (a group). So there are two possible ways to get a group parameter for a certain group, this is similar to the situation in table data (with its `field()` method):

```
>>> hdul[0].data.par(0)[8]
8.1
>>> hdul[0].data[8].par(0)
8.1
```

On the other hand, to modify a group parameter, we can either assign the new value directly (if accessing the row/group number last) or use the `setpar()` method (if accessing the row/group number first). The method `setpar()` is also needed for updating by name if the parameter is shared by more than one parameters:

```
>>> # Update group parameter when selecting the row (group) number
last
>>> hdul[0].data.par(0)[8] = 99.
>>> # Update group parameter when selecting the row (group) number
first
>>> hdul[0].data[8].setpar(0, 99.)  # or:
>>> hdul[0].data[8].setpar('abc', 99.)
>>> # Update group parameter by name when the name is shared by more
than
>>> # one parameters, the new value must be a tuple of constants or
>>> # sequences
>>> hdul[0].data[8].setpar('xyz', (2445729., 0.3))
>>> hdul[0].data[8:].par('xyz')
array([2.44572930e+06, 8.40000000e+01])
```

Data: Image Data

The image array of the data portion is accessible by the `data` attribute of the data object. A `numpy` array is returned:

```
>>> print(hdul[0].data.data[8])
[[[[120. 121. 122. 123. 124.]
   [125. 126. 127. 128. 129.]
   [130. 131. 132. 133. 134.]]]]
>>> hdul.close()
```

Creating a Random Access Group HDU

To create a Random Access Group HDU from scratch, use **GroupData** to encapsulate the data into the group data structure, and use **GroupsHDU** to create the HDU itself.

Example

To create a Random Access Group HDU:

```
>>> # Create the image arrays. The first dimension is the number of
groups.
>>> imdata = np.arange(150.0).reshape(10, 1, 1, 3, 5)
>>> # Next, create the group parameter data, we'll have two
parameters.
>>> # Note that the size of each parameter's data is also the number
of
>>> # groups.
>>> # A parameter's data can also be a numeric constant.
>>> pdata1 = np.arange(10) + 0.1
>>> pdata2 = 42
>>> # Create the group data object, put parameter names and parameter
data
>>> # in lists assigned to their corresponding arguments.
>>> # If the data type (bitpix) is not specified, the data type of
the
>>> # image will be used.
>>> x = fits.GroupData(imdata, bitpix=-32,
...                     parnames=['abc', 'xyz', 'xyz'],
...                     pardata=[pdata1, pdata2, pdata2])
>>> # Now, create the GroupsHDU and write to a FITS file.
>>> hdu = fits.GroupsHDU(x)
>>> hdu.writeto('test_group.fits')
>>> hdu.header
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                  -32 / array data type
NAXIS   =                    5 / number of array dimensions
NAXIS1  =                    0
NAXIS2  =                    5
NAXIS3  =                    3
NAXIS4  =                    1
NAXIS5  =                    1
EXTEND  =                    T
GROUPS  =                    T / has groups
PCOUNT  =                    3 / number of parameters
GCOUNT  =                   10 / number of groups
PTYPE1  = 'abc     '
PTYPE2  = 'xyz     '
PTYPE3  = 'xyz     '
>>> data = hdu.data
>>> hdu.data
GroupData([ (0.1       , 42., 42., [[[[  0.,   1.,   2.,   3.,   4.],
[  5.,   6.,   7.,   8.,   9.], [ 10.,  11.,  12.,  13.,  14.]]]]),
           (1.10000002, 42., 42., [[[[ 15.,  16.,  17.,  18.,  19.],
[ 20.,  21.,  22.,  23.,  24.], [ 25.,  26.,  27.,  28.,  29.]]]]),
           (2.0999999 , 42., 42., [[[[ 30.,  31.,  32.,  33.,  34.],
[ 35.,  36.,  37.,  38.,  39.], [ 40.,  41.,  42.,  43.,  44.]]]]),
           (3.0999999 , 42., 42., [[[[ 45.,  46.,  47.,  48.,  49.],
[ 50.,  51.,  52.,  53.,  54.], [ 55.,  56.,  57.,  58.,  59.]]]]),
           (4.0999999 , 42., 42., [[[[ 60.,  61.,  62.,  63.,  64.],
[ 65.,  66.,  67.,  68.,  69.], [ 70.,  71.,  72.,  73.,  74.]]]]),
           (5.0999999 , 42., 42., [[[[ 75.,  76.,  77.,  78.,  79.],
[ 80.,  81.,  82.,  83.,  84.], [ 85.,  86.,  87.,  88.,  89.]]]]),
           (6.0999999 , 42., 42., [[[[ 90.,  91.,  92.,  93.,  94.],
[ 95.,  96.,  97.,  98.,  99.], [100., 101., 102., 103., 104.]]]]),
           (7.0999999 , 42., 42., [[[[105., 106., 107., 108., 109.],
[110., 111., 112., 113., 114.], [115., 116., 117., 118., 119.]]]]),
           (8.10000038, 42., 42., [[[[120., 121., 122., 123., 124.],
[125., 126., 127., 128., 129.], [130., 131., 132., 133., 134.]]]]),
           (9.10000038, 42., 42., [[[[135., 136., 137., 138., 139.],
[140., 141., 142., 143., 144.], [145., 146., 147., 148., 149.]]]])],
           dtype=(numpy.record, [('abc', '<f4'), ('xyz', '<f4'),
('_xyz', '<f4'), ('DATA', '<f4', (1, 1, 3, 5))]))
```

Compressed Image Data

A general technique has been developed for storing compressed image data in FITS binary tables. The principle used in this convention is to first divide the n-dimensional image into a rectangular grid of sub-images or ‘tiles’. Each tile is then compressed as a continuous block of data, and the resulting compressed byte stream is stored in a row of a variable length column in a FITS binary table. Several commonly used algorithms for compressing image tiles are supported. These include Gzip, Rice, IRAF Pixel List (PLIO), and Hcompress.

For more details, reference “A FITS Image Compression Proposal” from:

> https://www.adass.org/adass/proceedings/adass99/P2-42/

and “Registered FITS Convention, Tiled Image Compression Convention”:

> https://fits.gsfc.nasa.gov/registry/tilecompression.html

Compressed image data is accessed, in `astropy`, using the optional `astropy.io.fits.compression` module contained in a C shared library (compression.so). If an attempt is made to access an HDU containing compressed image data when the compression module is not available, the user is notified of the problem and the HDU is treated like a standard binary table HDU. This notification will only be made the first time compressed image data is encountered. In this way, the compression module is not required in order for `astropy` to work.

Header and Summary

In `astropy`, the header of a compressed image HDU appears to the user like any image header. The actual header stored in the FITS file is that of a binary table HDU with a set of special keywords, defined by the convention, to describe the structure of the compressed image. The conversion between binary table HDU header and image HDU header is all performed behind the scenes. Since the HDU is actually a binary table, it may not appear as a primary HDU in a FITS file.

Example

The content of the HDU header may be accessed using the `.header` attribute:

```
>>> filename =
fits.util.get_testdata_filepath('compressed_image.fits')

>>> hdul = fits.open(filename)
>>> hdul[1].header
XTENSION= 'IMAGE   '           / Image extension
BITPIX  =                   16 / data type of original image
NAXIS   =                    2 / dimension of original image
NAXIS1  =                   10 / length of original image axis
NAXIS2  =                   10 / length of original image axis
PCOUNT  =                    0 / number of parameters
GCOUNT  =                    1 / number of groups
```

The contents of the corresponding binary table HDU may be accessed using the hidden `._header` attribute. However, all user interface with the HDU header should be accomplished through the image header (the `.header` attribute):

```
>>> hdul[1]._header
XTENSION= 'BINTABLE'           / binary table extension
BITPIX  =                    8 / array data type
NAXIS   =                    2 / number of array dimensions
NAXIS1  =                    8 / width of table in bytes
NAXIS2  =                   10 / number of rows in table
PCOUNT  =                   60 / number of group parameters
GCOUNT  =                    1 / number of groups
TFIELDS =                    1 / number of fields in each row
```

```
TTYPE1  = 'COMPRESSED_DATA'    / label for field 1
TFORM1  = '1PB(6)  '           / data format of field: variable
length array
ZIMAGE  =                    T / extension contains compressed image
ZTENSION= 'IMAGE   '           / Image extension
ZBITPIX =                   16 / data type of original image
ZNAXIS  =                    2 / dimension of original image
ZNAXIS1 =                   10 / length of original image axis
ZNAXIS2 =                   10 / length of original image axis
ZPCOUNT =                    0 / number of parameters
ZGCOUNT =                    1 / number of groups
ZTILE1  =                   10 / size of tiles to be compressed
ZTILE2  =                    1 / size of tiles to be compressed
ZCMPTYPE= 'RICE_1  '           / compression algorithm
ZNAME1  = 'BLOCKSIZE'          / compression block size
ZVAL1   =                   32 / pixels per block
ZNAME2  = 'BYTEPIX '           / bytes per pixel (1, 2, 4, or 8)
ZVAL2   =                    2 / bytes per pixel (1, 2, 4, or 8)
EXTNAME = 'COMPRESSED_IMAGE'   / name of this binary table extension
```

The contents of the HDU can be summarized by using either the **info()** convenience function or method:

```
>>> fits.info(filename)
Filename: ...compressed_image.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU       4   ()
  1  COMPRESSED_IMAGE    1 CompImageHDU      7   (10, 10)   int16

>>> hdul.info()
Filename: ...compressed_image.fits
No.    Name      Ver    Type      Cards   Dimensions   Format
  0  PRIMARY       1 PrimaryHDU       4   ()
  1  COMPRESSED_IMAGE    1 CompImageHDU      7   (10, 10)   int16
```

Data

As with the header, the data of a compressed image HDU appears to the user as standard uncompressed image data. The actual data is stored in the FITS file as Binary Table data containing at least one column (COMPRESSED_DATA). Each row of this variable length column contains the byte stream that was generated as a result of compressing the corresponding image tile. Several optional columns may also appear. These include UNCOMPRESSED_DATA to hold the uncompressed pixel values for tiles that cannot be compressed, ZSCALE and ZZERO to hold the linear scale factor and zero point offset which may be needed to transform the raw uncompressed values back to the original image pixel values, and ZBLANK to hold the integer value used to represent undefined pixels (if any) in the image.

Example

The contents of the uncompressed HDU data may be accessed using the `.data` attribute:

```
>>> hdul[1].data
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
       [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
       [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
       [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
       [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
       [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]], dtype=int16)
>>> hdul.close()
```

The compressed data can be accessed via the `.compressed_data` attribute, but this rarely needs be accessed directly. It may be useful for performing direct copies of the compressed data without needing to decompress it first.

Creating a Compressed Image HDU

To create a compressed image HDU from scratch, construct a **`CompImageHDU`** object from an uncompressed image data array and its associated image header. From there, the HDU can be treated just like any other image HDU.

Example

To create a compressed image HDU:

```
>>> imageData = np.arange(100).astype('i2').reshape(10, 10)
>>> imageHeader = fits.Header()
>>> hdu = fits.CompImageHDU(imageData, imageHeader)
>>> hdu.writeto('compressed_image.fits')
```

The API documentation for the **`CompImageHDU`** initializer method describes the possible options for constructing a **`CompImageHDU`** object.

Executable Scripts

`astropy` installs a couple of useful utility programs on your system that are built with `astropy`.

fitsinfo

`fitsinfo` is a command-line script based on astropy.io.fits for printing a summary of the HDUs in one or more FITS files(s) to the standard output.

Example usage of `fitsinfo`:

1. Print a summary of the HDUs in a FITS file:

```
$ fitsinfo filename.fits

Filename: filename.fits
No.    Name         Type      Cards   Dimensions   Format
0    PRIMARY     PrimaryHDU     138   ()
1    SCI         ImageHDU        61   (800, 800)   int16
2    SCI         ImageHDU        61   (800, 800)   int16
3    SCI         ImageHDU        61   (800, 800)   int16
4    SCI         ImageHDU        61   (800, 800)   int16
```

2. Print a summary of HDUs of all the FITS files in the current directory:

```
$ fitsinfo *.fits
```

fitsheader

`fitsheader` is a command line script based on astropy.io.fits for printing the header(s) of one or more FITS file(s) to the standard output in a human-readable format.

Example uses of fitsheader:

1. Print the header of all the HDUs of a .fits file:

```
$ fitsheader filename.fits
```

2. Print the header of the third and fifth HDU extension:

```
$ fitsheader --extension 3 --extension 5 filename.fits
```

3. Print the header of a named extension, e.g. select the HDU containing keywords EXTNAME='SCI' and EXTVER='2':

```
$ fitsheader --extension "SCI,2" filename.fits
```

4. Print only specific keywords:

```
$ fitsheader --keyword BITPIX --keyword NAXIS filename.fits
```

5. Print keywords NAXIS, NAXIS1, NAXIS2, etc using a wildcard:

```
$ fitsheader --keyword NAXIS* filename.fits
```

6. Dump the header keywords of all the files in the current directory into a machine-readable csv file:

```
$ fitsheader --table ascii.csv *.fits > keywords.csv
```

7. Specify hierarchical keywords with the dotted or spaced notation:

```
$ fitsheader --keyword ESO.INS.ID filename.fits
$ fitsheader --keyword "ESO INS ID" filename.fits
```

8. Compare the headers of different fites files, following ESO's `fitsort` format:

```
$ fitsheader --fitsort --extension 0 --keyword ESO.INS.ID *.fits
```

9. Same as above, sorting the output along a specified keyword:

```
$ fitsheader -f DATE-OBS -e 0 -k DATE-OBS -k ESO.INS.ID *.fits
```

Note that compressed images (HDUs of type **`CompImageHDU`**) really have two headers: a real BINTABLE header to describe the compressed data, and a fake IMAGE header representing the image that was compressed. Astropy returns the latter by default. You must supply the `--compressed` option if you require the real header that describes the compression.

With Astropy installed, please run `fitsheader --help` to see the full usage documentation.

fitscheck

`fitscheck` is a command line script based on astropy.io.fits for verifying and updating the CHECKSUM and DATASUM keywords of .fits files. `fitscheck` can also detect and often fix other FITS standards violations. `fitscheck` facilitates re-writing the non-standard checksums originally generated by astropy.io.fits with standard checksums which will interoperate with CFITSIO.

`fitscheck` will refuse to write new checksums if the checksum keywords are missing or their values are bad. Use `--force` to write new checksums regardless of whether or not they currently exist or pass. Use `--ignore-missing` to tolerate missing checksum keywords without comment.

Example uses of fitscheck:

1. Add checksums:

   ```
   $ fitscheck --write *.fits
   ```

2. Write new checksums, even if existing checksums are bad or missing:

   ```
   $ fitscheck --write --force *.fits
   ```

3. Verify standard checksums and FITS compliance without changing the files:

   ```
   $ fitscheck --compliance *.fits
   ```

4. Only check and fix compliance problems, ignoring checksums:

   ```
   $ fitscheck --checksum none --compliance --write *.fits
   ```

5. Verify standard interoperable checksums:

   ```
   $ fitscheck *.fits
   ```

6. Delete checksum keywords:

   ```
   $ fitscheck --checksum remove --write *.fits
   ```

With `astropy` installed, please run `fitscheck --help` to see the full program usage documentation.

fitsdiff

`fitsdiff` provides a thin command-line wrapper around the **`FITSDiff`** interface. It outputs the report from a **`FITSDiff`** of two FITS files, and like common diff-like commands returns a 0 status code if no differences were found, and 1 if differences were found:

With `astropy` installed, please run `fitsdiff --help` to see the full program usage documentation.

Miscellaneous Features

This section describes some of the miscellaneous features of **astropy.io.fits**.

Differs

The **astropy.io.fits.diff** module contains several facilities for generating and reporting the differences between two FITS files, or two components of a FITS file.

The **FITSDiff** class can be used to generate and represent the differences between either two FITS files on disk, or two existing **HDUList** objects (or some combination thereof).

Likewise, the **HeaderDiff** class can be used to find the differences just between two **Header** objects. Other available differs include **HDUDiff**, **ImageDataDiff**, **TableDataDiff**, and **RawDataDiff**.

Each of these classes are instantiated with two instances of the objects that they diff. The returned diff instance has a number of attributes starting with `.diff_` that describe differences between the two objects.

Example

The **HeaderDiff** class can be used to find the differences between two **Header** objects like so:

```
>>> from astropy.io import fits
>>> header1 = fits.Header([('KEY_A', 1), ('KEY_B', 2)])
>>> header2 = fits.Header([('KEY_A', 3), ('KEY_C', 4)])
>>> diff = fits.diff.HeaderDiff(header1, header2)
>>> diff.identical
False
>>> diff.diff_keywords
(['KEY_B'], ['KEY_C'])
>>> diff.diff_keyword_values
defaultdict(..., {'KEY_A': [(1, 3)]})
```

See the API documentation for details on the different differ classes.

Command-Line Utilities

For convenience, several of `astropy`'s sub-packages install utility programs on your system which allow common tasks to be performed without having to open a Python interpreter. These utilities include:

- **`fitsheader`**: prints the headers of a FITS file.
- **`fitscheck`**: verifies and optionally rewrites the CHECKSUM and DATASUM keywords of a FITS file.
- fitsdiff: compares two FITS files and reports the differences.
- Scripts: converts FITS images to bitmaps, including scaling and stretching.
- wcslint: checks the WCS keywords in a FITS file for compliance against the standards.

Other Information

astropy.io.fits FAQ

Contents

General Questions

What is PyFITS and how does it relate to astropy?

PyFITS is a library written in, and for use with the Python programming language for reading, writing, and manipulating FITS formatted files. It includes a high-level interface to FITS headers with the ability for high- and low-level manipulation of headers, and it supports reading image and table data as Numpy arrays. It also supports more obscure and nonstandard formats found in some FITS files.

The **`astropy.io.fits`** module is identical to PyFITS but with the names changed. When the development of `astropy` began, it was clear that one of the core requirements would be a FITS reader. Rather than starting from scratch, PyFITS — being the most flexible FITS reader available for Python — was ported into `astropy`. There are plans to gradually phase out PyFITS as a stand-alone module and deprecate it in favor of **`astropy.io.fits`**. See more about this in the next question.

Although PyFITS is written mostly in Python, it includes an optional module written in C that is required to read/write compressed image data. However, the rest of PyFITS functions without this extension module.

What is the development status of PyFITS?

PyFITS was written and maintained by the Science Software Branch at the Space Telescope Science Institute, and is licensed by AURA under a 3-clause BSD license.

It is now exclusively developed as a component of `astropy` (**`astropy.io.fits`**) rather than as a stand-alone module. There are a few reasons for this: The first is simply to reduce development effort; the overhead of maintaining both PyFITS *and* **`astropy.io.fits`** in separate code bases is nontrivial. The second is that there are many features of `astropy` (units, tables, etc.) from which the **`astropy.io.fits`** module can benefit greatly. Since PyFITS is already integrated into `astropy`, it makes more sense to continue development there rather than make `astropy` a dependency of PyFITS.

PyFITS' past primary developer and active maintainer was Erik Bray. There is a GitHub project for PyFITS, but PyFITS is not actively developed anymore so patches and issue reports should be posted on the Astropy issue tracker.

The current (and last) stable release is 3.4.0.

Usage Questions

Something did not work as I expected. Did I do something wrong?

Possibly. But if you followed the documentation and things still did not work as expected, it is entirely possible that there is a mistake in the documentation, a bug in the code, or both. So feel free to report it as a bug. There are also many, many corner cases in FITS files, with new ones discovered almost every week. **`astropy.io.fits`** is always improving, but does not support all cases perfectly. There are some features of the FITS format (scaled data, for example) that are difficult to support correctly and can sometimes cause unexpected behavior.

For the most common cases, however, such as reading and updating FITS headers, images, and tables, **`astropy.io.fits`** is very stable and well-tested. Before every `astropy` release it is ensured that all of its tests pass on a variety of platforms, and those tests cover the majority of use cases (until new corner cases are discovered).

astropy crashed and output a long string of code. What do I do?

This listing of code is what is known as a stack trace (or in Python parlance a "traceback"). When an unhandled exception occurs in the code causing the program to end, this is a way of displaying where the exception occurred and the path through the code that led to it.

As `astropy` is meant to be used as a piece in other software projects, some exceptions raised by `astropy` are by design. For example, one of the most common exceptions is a **`KeyError`** when an attempt is made to read the value of a nonexistent keyword in a header:

```
>>> from astropy.io import fits
>>> h = fits.Header()
>>> h['NAXIS']
Traceback (most recent call last):
    ...
KeyError: "Keyword 'NAXIS' not found."
```

This indicates that something was looking for a keyword called "NAXIS" that does not exist. If an error like this occurs in some other software that uses `astropy`, it may indicate a bug in that software, in that it expected to find a keyword that did not exist in a file.

Most "expected" exceptions will output a message at the end of the traceback giving some idea of why the exception occurred and what to do about it. The more vague and mysterious the error message in an exception appears, the more likely that it was caused by a bug in `astropy`. So if you are getting an exception and you really do not know why or what to do about it, feel free to report it as a bug.

Why does opening a file work in CFITSIO, ds9, etc., but not in `astropy`?

As mentioned elsewhere in this FAQ, there are many unusual corner cases when dealing with FITS files. It is possible that a file should work, but is not supported due to a bug. Sometimes it is even possible for a file to work in an older version of `astropy`, but not a newer version due to a regression that has not been tested for yet.

Another problem with the FITS format is that, as old as it is, there are many conventions that appear in files from certain sources that do not meet the FITS standard. And yet they are so commonplace that it is necessary to support them in any FITS readers. CONTINUE cards are one such example. There are nonstandard conventions supported by `astropy` that are not supported by CFITSIO and possibly vice versa. You may have hit one of those cases.

If `astropy` is having trouble opening a file, a good way to rule out whether not the problem is with `astropy` is to run the file through the fitsverify program. For smaller files you can even use the online FITS verifier. These use CFITSIO under the hood, and should give a good indication of whether or not there is something erroneous about the file. If the file is malformatted, fitsverify will output errors and warnings.

If fitsverify confirms no problems with a file, and `astropy` is still having trouble opening it (especially if it produces a traceback), then it is possible there is a bug in `astropy`.

How do I turn off the warning messages `astropy` outputs to my console?

`astropy` uses Python's built-in warnings subsystem for informing about exceptional conditions in the code that are recoverable, but that the user may want to be informed of. One of the most common warnings in **`astropy.io.fits`** occurs when updating a header value in such a way that the comment must be truncated to preserve space:

```
Card is too long, comment is truncated.
```

Any console output generated by `astropy` can be assumed to be from the warnings subsystem. See Astropy's documentation on the Python warnings system for more information on how to control and quiet warnings.

What convention does `astropy` use for indexing, such as of image coordinates?

All arrays and sequences in `astropy` use a zero-based indexing scheme. For example, the first keyword in a header is `header[0]`, not `header[1]`. This is in accordance with Python itself, as well as C, on which Python is based.

This may come as a surprise to veteran FITS users coming from IRAF, where 1-based indexing is typically used, due to its origins in Fortran.

Likewise, the top-left pixel in an N x N array is `data[0,0]`. The indices for 2-dimensional arrays are row-major order, in that the first index is the row number, and the second index is the column number. Or put in terms of axes, the first axis is the y-axis, and the second axis is the x-axis. This is the opposite of column-major order, which is used by Fortran and hence FITS. For example, the second index refers to the axis specified by NAXIS1 in the FITS header.

In general, for N-dimensional arrays, row-major orders means that the right-most axis is the one that varies the fastest while moving over the array data linearly. For example, the 3-dimensional array:

```
[[[1, 2],
  [3, 4]],
 [[5, 6],
  [7, 8]]]
```

is represented linearly in row-major order as:

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

Since 2 immediately follows 1, you can see that the right-most (or inner-most) axis is the one that varies the fastest.

The discrepancy in axis-ordering may take some getting used to, but it is a necessary evil. Since most other Python and C software assumes row-major ordering, trying to enforce column-major ordering in arrays returned by `astropy` is likely to cause more difficulties than it is worth.

How do I open a very large image that will not fit in memory?

`astropy.io.fits.open` has an option to access the data portion of an HDU by memory mapping using mmap. In `astropy` this is used by default.

What this means is that accessing the data as in the example above only reads portions of the data into memory on demand. For example, if we request just a slice of the image, such as `hdul[0].data[100:200]`, then only rows 100-200 will be read into memory. This happens transparently, as though the entire image were already in memory. This works the same way for tables. For most cases this is your best bet for working with large files.

To ensure use of memory mapping, add the `memmap=True` argument to fits.open. Likewise, using `memmap=False` will force data to be read entirely into memory.

The default can also be controlled through a configuration option called `USE_MEMMAP`. Setting this to `0` will disable mmap by default.

Unfortunately, memory mapping does not currently work as well with scaled image data, where BSCALE and BZERO factors need to be applied to the data to yield physical values. Currently this requires enough memory to hold the entire array, though this is an area that will see improvement in the future.

An alternative, which currently only works for image data (that is, non-tables) is the sections interface. It is largely replaced by the better support for mmap, but may still be useful on systems with more limited virtual memory space, such as on 32-bit systems. Support for scaled image data is flaky with sections too, though that will be fixed. See the documentation on image sections for more details on using this interface.

How can I create a very large FITS file from scratch?

See Create a very large FITS file from scratch.

For creating very large tables, this method may also be used, though it can be difficult to determine ahead of time how many rows a table will need. In general, use of the **`astropy.io.fits`** module is currently discouraged for the creation and manipulation of large tables. The FITS format itself is not designed for efficient on-disk or in-memory manipulation of table structures. For large, heavy-duty table data it might be better too look into using HDF5 through the PyTables library. The Astropy Table interface can provide an abstraction layer between different on-disk table formats as well (for example, for converting a table between FITS and HDF5).

PyTables makes use of NumPy under the hood, and can be used to write binary table data to disk in the same format required by FITS. It is then possible to serialize your table to the FITS format for distribution. At some point this FAQ might provide an example of how to do this.

How do I create a multi-extension FITS file from scratch?

See Create a multi-extension FITS (MEF) file from scratch.

Why is an image containing integer data being converted unexpectedly to

floats?

If the header for your image contains nontrivial values for the optional BSCALE and/or BZERO keywords (that is, BSCALE != 1 and/or BZERO != 0), then the raw data in the file must be rescaled to its physical values according to the formula:

```
physical_value = BZERO + BSCALE * array_value
```

As BZERO and BSCALE are floating point values, the resulting value must be a float as well. If the original values were 16-bit integers, the resulting values are single-precision (32-bit) floats. If the original values were 32-bit integers, the resulting values are double-precision (64-bit floats).

This automatic scaling can easily catch you off guard if you are not expecting it, because it does not happen until the data portion of the HDU is accessed (to allow for things like updating the header without rescaling the data). For example:

```
>>> fits_scaledimage_filename =
fits.util.get_testdata_filepath('scale.fits')

>>> hdul = fits.open(fits_scaledimage_filename)
>>> image = hdul[0]
>>> image.header['BITPIX']
16
>>> image.header['BSCALE']
0.045777764213996
>>> data = image.data  # Read the data into memory
>>> data.dtype.name    # Got float32 despite BITPIX = 16 (16-bit int)
'float32'
>>> image.header['BITPIX']  # The BITPIX will automatically update
too
-32
>>> 'BSCALE' in image.header  # And the BSCALE keyword removed
False
```

The reason for this is that once a user accesses the data they may also manipulate it and perform calculations on it. If the data were forced to remain as integers, a great deal of precision is lost. So it is best to err on the side of not losing data, at the cost of causing some confusion at first.

If the data must be returned to integers before saving, use the **`scale`** method:

```
>>> image.scale('int32')
>>> image.header['BITPIX']
32
>>> hdul.close()
```

Alternatively, if a file is opened with `mode='update'` along with the `scale_back=True` argument, the original BSCALE and BZERO scaling will be automatically reapplied to the data before saving. Usually this is not desirable, especially when converting from floating point values back to unsigned integer values. But this may be useful in cases where the raw data needs to be modified corresponding to changes in the physical values.

To prevent rescaling from occurring at all (which is good for updating headers — even if you do not intend for the code to access the data, it is good to err on the side of caution here), use the `do_not_scale_image_data` argument when opening the file:

```
>>> hdul = fits.open(fits_scaledimage_filename,
do_not_scale_image_data=True)
>>> image = hdul[0]
>>> image.data.dtype.name
'int16'
>>> hdul.close()
```

Why am I losing precision when I assign floating point values in the header?

The FITS standard allows two formats for storing floating point numbers in a header value. The “fixed” format requires the ASCII representation of the number to be in bytes 11 through 30 of the header card, and to be right-justified. This leaves a standard number of characters for any comment string.

The fixed format is not wide enough to represent the full range of values that can be stored in a 64-bit float with full precision. So FITS also supports a “free” format in which the ASCII representation can be stored anywhere, using the full 70 bytes of the card (after the keyword).

Currently `astropy` only supports writing fixed format (it can read both formats), so all floating point values assigned to a header are stored in the fixed format. There are plans to add support for more flexible formatting.

In the meantime, it is possible to add or update cards by manually formatting the card image from a string, as it should appear in the FITS file:

```
>>> c = fits.Card.fromstring('FOO     = 1234567890.123456789')
>>> h = fits.Header()
>>> h.append(c)
>>> h
```

```
FOO     = 1234567890.123456789
```

As long as you do not assign new values to ‘FOO’ via `h['FOO'] = 123`, will maintain the header value exactly as you formatted it (as long as it is valid according to the FITS standard).

Why is reading rows out of a FITS table so slow?

Underlying every table data array returned by **`astropy.io.fits`** is a `numpy` **`recarray`** which is a `numpy` array type specifically for representing structured array data (i.e., a table). As with normal image arrays, `astropy` accesses the underlying binary data from the FITS file via mmap (see the question “What performance differences are there between astropy.io.fits and fitsio?” for a deeper explanation of this). The underlying mmap is then exposed as a **`recarray`** and in general this is a very efficient way to read the data.

However, for many (if not most) FITS tables it is not all that simple. For many columns there are conversions that have to take place between the actual data that is “on disk” (in the FITS file) and the data values that are returned to the user. For example, FITS binary tables represent boolean values differently from how `numpy` expects them to be represented, “Logical” columns need to be converted on the fly to a format `numpy` (and hence the user) can understand. This issue also applies to data that is linearly scaled via the `TSCALn` and `TZEROn` header keywords.

Supporting all of these “FITS-isms” introduces a lot of overhead that might not be necessary for all tables, but are still common nonetheless. That is not to say it cannot be faster even while supporting the peculiarities of FITS — CFITSIO, for example, supports all of the same features but is orders of magnitude faster. `astropy` could do much better here too, and there are many known issues causing slowdown. There are plenty of opportunities for speedups, and patches are welcome. In the meantime, for high-performance applications with FITS tables some users might find the `fitsio` library more to their liking.

I am opening many FITS files in a loop and getting OSError: Too many open files

Say you have some code like:

```
from astropy.io import fits

for filename in filenames:
    with fits.open(filename) as hdul:
        for hdu in hdul:
            hdu_data = hdul.data
            # Do some stuff with the data
```

The details may differ, but the qualitative point is that the data to many HDUs and/or FITS files are being accessed in a loop. This may result in an exception like:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
OSError: [Errno 24] Too many open files: 'my_data.fits'
```

As explained in the note on working with large files, because `astropy` uses mmap by default to read the data in a FITS file, even if you correctly close a file with HDUList.close a handle is kept open to that file so that the memory-mapped data array can still continue to be read transparently.

The way `numpy` supports mmap is such that the file mapping is not closed until the overlying **`ndarray`** object has no references to it and is freed memory. However, when looping over a large number of files (or even just HDUs) rapidly, this may not happen immediately. Or in some cases if the HDU object persists, the data array attached to it may persist too. The recommended workaround is to *manually* delete the `.data` attribute on the HDU object so that the **`ndarray`** reference is freed and the mmap can be closed:

```
from astropy.io import fits

for filename in filenames:
    with fits.open(filename) as hdul:
        for hdu in hdul:
            hdu_data = hdul.data
            # Do some stuff with the data
            # ...
            # Don't need the data anymore; delete all references to
it
            # so that it can be garbage collected
            del hdu_data
            del hdu.data
```

In some extreme cases files are opened and closed fast enough that Python's garbage collector does not free them (and hence free the file handles) often enough. To mitigate this, your code can manually force a garbage collection by calling **`gc.collect()`** at the end of the loop.

In a future release it will be more convenient to automatically perform this sort of cleanup when closing FITS files, where needed.

Using header['NAXIS2'] += 1 does not add another row to my Table

`NAXIS` and similar keywords are FITS *structural* keywords and should not be modified by the user. They are automatically updated by **`astropy.io.fits`** when checking the validity of the data and headers. See Structural Keywords for more information.

To add rows to a table, you can modify the actual data.

Comparison with Other FITS Readers

What is the difference between astropy.io.fits and fitsio?

The **`astropy.io.fits`** module (originally PyFITS) is a “pure Python” FITS reader in that all of the code for parsing the FITS file format is in Python, though `numpy` is used to provide access to the FITS data via the **`ndarray`** interface. **`astropy.io.fits`** currently also accesses the CFITSIO to support the FITS Tile Compression convention, but this feature is optional. It does not use CFITSIO outside of reading compressed images.

fitsio, on the other hand, is a Python wrapper for the CFITSIO library. All of the heavy lifting of reading the FITS format is handled by CFITSIO, while `fitsio` provides a better way to use object-oriented API, including providing a `numpy` interface to FITS files read from CFITSIO. Much of it is written in C (to provide the interface between Python and CFITSIO), and the rest is in Python. The Python end mostly provides the documentation and user-level API.

Because `fitsio` wraps CFITSIO it inherits most of its strengths and weaknesses, though it has an added strength of providing a more convenient API than if one were to use CFITSIO directly.

Why did Astropy adopt PyFITS as its FITS reader instead of fitsio?

When the Astropy Project was first started it was clear from the start that one of its core components should be a submodule for reading and writing FITS files, as many other components would be likely to depend on this functionality. At the time, the `fitsio` package was in its infancy (it goes back to roughly 2011) while PyFITS had already been established (going back to before the year 2000). It was already a mature package with support for the vast majority of FITS files found in the wild, including outdated formats such as “Random Groups” FITS files still used extensively in the radio astronomy community.

Although many aspects of PyFITS’ interface have evolved over the years, much of it has also remained the same, and is already familiar to astronomers working with FITS files in Python. Most of if not all existing training materials were also based around PyFITS. PyFITS was developed at STScI, which also put forward significant resources to develop Astropy, with an eye toward integrating Astropy into STScI’s own software stacks. As most of the Python software at STScI uses PyFITS, it was the only practical choice for making that transition.

Finally, although CFITSIO (and by extension `fitsio`) can read any FITS files that conform to the FITS standard, it does not support all of the nonstandard conventions that have been added to FITS files in the wild. While it does have some support for some of these conventions (such as CONTINUE cards and, to a limited extent, HIERARCH cards), it is not easy to add support for other conventions to a large and complex C codebase.

PyFITS' object-oriented design makes supporting nonstandard conventions somewhat easier in most cases, and as such PyFITS can be more flexible in the types of FITS files it can read and return *useful* data from. This includes better support for files that fail to meet the FITS standard, but still contain useful data that should be readable enough to correct any violations of the FITS standard. For example, a common error in non-English speaking regions is to insert non-ASCII characters into FITS headers. This is not a valid FITS file, but should still be readable in some sense. Supporting structural errors such as this is more difficult in CFITSIO which assumes a more rigid structure.

What performance differences are there between astropy.io.fits and fitsio?

There are two main performance areas to look at: reading/parsing FITS headers and reading FITS data (image-like arrays as well as tables).

In the area of headers, `fitsio` is significantly faster in most cases. This is due in large part to the (almost) pure C implementation (due to the use of CFITSIO), but also due to fact that it is more rigid and does not support as many local conventions and other special cases as **`astropy.io.fits`** tries to support in its pure Python implementation.

That said, the difference is small and only likely to be a bottleneck either when opening files containing thousands of HDUs, or reading the headers out of thousands of FITS files in succession (in either case the difference is not even an order of magnitude).

Where data is concerned the situation is a little more complicated, and requires some understanding of how **`astropy.io.fits`** is implemented versus CFITSIO and `fitsio`. First, it is important to understand how they differ in terms of memory management.

`astropy.io.fits` uses mmap, by default, to provide access to the raw binary data in FITS files. Mmap is a system call (or in most cases these days a wrapper in your libc for a lower-level system call) which allows user-space applications to essentially do the same thing your OS is doing when it uses a pagefile (swap space) for virtual memory: it allows data in a file on disk to be paged into physical memory one page (or in practice usually several pages) at a time on an as-needed basis. These cached pages of the file are also accessible from all processes on the system, so multiple processes can read from the same file with little additional overhead. In the case of reading over all of the data in the file, the performance difference between using mmap versus reading the entire data into physical memory at once can vary widely between systems, hardware, and depending on what else is happening on the system at the moment, but mmap is almost always going to be better.

In principle, it requires more overhead since accessing each page will result in a page fault and the system requires more requests to the disk. But in practice, the OS will optimize this pretty aggressively, especially for the most common case of sequential access — also in reality, reading the entire thing into memory is still going to result in a whole lot of page faults too. For random access, having all of the data in physical memory is always going to be best, though with mmap it is usually going to be pretty good too. (Most users do not normally access all of the data in a file in a totally random order — usually a few sections of it will be accessed most frequently, so the OS will keep those pages in physical memory as best it can.) For the most general case of reading FITS files (or most large data on disk) this is therefore the best choice, especially for casual users, and is hence enabled by default.

CFITSIO/ `fitsio` , on the other hand, does not assume the existence of technologies like mmap and page caching. Thus it implements its own LRU cache of I/O buffers that store sections of FITS files read from disk in memory in FITS' famous 2880 byte chunk size. The I/O buffers are used heavily in particular for keeping the headers in memory. Though for large data reads (for example, reading an entire image from a file), it *does* bypass the cache and instead does a read directly from disk into a user-provided memory buffer.

However, even when CFITSIO reads direct from the file, this is still largely less efficient than using mmap. Normally when your OS reads a file from disk, it caches as much of that read as it can in physical memory (in its page cache) so that subsequent access to those same pages does not require a subsequent expensive disk read. This happens when using mmap too, since the data has to be copied from disk into RAM at some point. The difference is that when using mmap to access the data, the program is able to read that data *directly* out of the OS's page cache (as long as it is only being read). On the other hand, when reading data from a file into a local buffer such as with fread(), the data is first read into the page cache (if not already present) and then copied from the page cache into the local buffer. So every read performs at least one additional memory copy per page read (requiring twice as much physical memory, and possibly lots of paging if the file is large and pages need to dropped from the cache).

The user API for CFITSIO usually works by having the user allocate a memory buffer large enough to hold the image/table they want to read (or at least the section they are interested in). There are some helper functions for determining the appropriate amount of space to allocate. Then you pass in a pointer to your buffer and CFITSIO handles all of the reading (usually using the process described above), and copies the results into your user buffer. For large reads, it reads directly from the file into your buffer, though if the data needs to be scaled it makes a stop in CFITSIO’s own buffer first, then writes the rescaled values out to the user buffer (if rescaling has been requested). Regardless, this means that if your program wishes to hold an entire image in memory at once it will use as much RAM as the size of the data. For most applications it is better (and sufficient) to work on smaller sections of the data, but this requires extra complexity. Using mmap on the other hand makes managing this complexity more efficient.

An informal test demonstrates this difference. This test was performed on four simple FITS images (one of which is a cube) of dimensions 256x256, 1024x1024, 4096x4096, and 256x1024x1024. Each image was generated before the test and filled with randomized 64-bit floating point values. A similar test was performed using both `astropy.io.fits` and `fitsio`. A handle to the FITS file is opened using each library’s basic semantics, and then the entire data array of the files is copied into a temporary array in memory (for example, if we were blitting the image to a video buffer). For `astropy` the test is written:

```
def read_test_astropy(filename):
    with fits.open(filename, memmap=True) as hdul:
        data = hdul[0].data
        c = data.copy()
```

The test was timed in IPython on a Linux system with kernel version 2.6.32, a 6-core Intel Xeon X5650 CPU clocked at 2.67 GHz per core, and 11.6 GB of RAM using:

```
for filename in filenames:
    print(filename)
    %timeit read_test_astropy(filename)
```

where `filenames` is just a list of the aforementioned generated sample files. The results were:

```
256x256.fits
1000 loops, best of 3: 1.28 ms per loop
1024x1024.fits
100 loops, best of 3: 4.24 ms per loop
4096x4096.fits
10 loops, best of 3: 60.6 ms per loop
256x1024x1024.fits
1 loops, best of 3: 1.15 s per loop
```

For `fitsio` the test was:

```
def read_test_fitsio(filename):
    with fitsio.FITS(filename) as f:
        data = f[0].read()
        c = data.copy()
```

This was also run in a loop over all of the sample files, producing the results:

```
256x256.fits
1000 loops, best of 3: 476 µs per loop
1024x1024.fits
100 loops, best of 3: 12.2 ms per loop
4096x4096.fits
10 loops, best of 3: 136 ms per loop
256x1024x1024.fits
1 loops, best of 3: 3.65 s per loop
```

It should be made clear that the sample files were rewritten with new random data between the `astropy` test and the fitsio test, so they were not reading the same data from the OS's page cache. Fitsio was much faster on the small (256x256) image because in that case the time is dominated by parsing the headers. As already explained, this is much faster in CFITSIO. However, as the data size goes up and the header parsing no longer dominates the time, **`astropy.io.fits`** using mmap is roughly twice as fast. This discrepancy is almost entirely due to it requiring roughly half as many in-memory copies to read the data, as explained earlier. That said, more extensive benchmarking could be very interesting.

This is also not to say that **`astropy.io.fits`** does better in all cases. There are some cases where it is currently blown away by fitsio. See the subsequent question.

Why is fitsio so much faster than `astropy` at reading tables?

In many cases it is not: there is either no difference, or it may be a little faster in `astropy` depending on what you are trying to do with the table and what types of columns or how many columns the table has. There are some cases, however, where `fitsio` can be radically faster, mostly for reasons explained above in "Why is reading rows out of a FITS table so slow?"

In principle a table is no different from, say, an array of pixels. But instead of pixels each element of the array is some kind of record structure (for example, two floats, a boolean, and a 20-character string field). Just as a 64-bit float is an 8 byte record in an array, a row in such a table can be thought of as a 37 byte (in the case of the previous example) record in a 1D array of rows. So in principle everything that was explained in the answer to the question “What performance differences are there between astropy.io.fits and fitsio?” applies just as well to tables as it does to any other array.

However, FITS tables have many additional complexities that sometimes preclude streaming the data directly from disk, and instead require transformation from the on-disk FITS format to a format more immediately useful to the user. A common example is how FITS represents boolean values in binary tables. Another significantly more complicated example, is variable length arrays.

As explained in “Why is reading rows out of a FITS table so slow?”, **`astropy.io.fits`** does not currently handle some of these cases as efficiently as it could, in particular in cases where a user only wishes to read a few rows out of a table. Fitsio, on the other hand, has a better interface for copying one row at a time out of a table and performing the necessary transformations on that row *only*, rather than on the entire column or columns that the row is taken from. As such, for many cases `fitsio` gets much better performance and should be preferred for many performance-critical table operations.

Fitsio also exposes a microlanguage (implemented in CFITSIO) for making efficient SQL-like queries of tables (single tables only though — no joins or anything like that). This format, described in the CFITSIO documentation can in some cases perform more efficient selections of rows than might be possible with `numpy` alone, which requires creating an intermediate mask array in order to perform row selection.

Header Interface Transition Guide

Note

This guide was originally included with the release of PyFITS 3.1, and still references PyFITS in many places, though the examples have been updated for `astropy.io.fits`. It is still useful here for informational purposes, though Astropy has always used the PyFITS 3.1 Header interface.

PyFITS v3.1 included an almost complete rewrite of the **`Header`** interface. Although the new interface is largely compatible with the old interface (whether due to similarities in the design, or backwards-compatibility support), there are enough differences that a full explanation of the new interface is merited.

Background

Prior to 3.1, PyFITS users interacted with FITS headers by way of three different classes: `Card`, `CardList`, and `Header`.

The Card class represents a single header card with a keyword, value, and comment. It also contains all of the machinery for parsing FITS header cards, given the 80-character string, or “card image” read from the header.

The CardList class is actually a subclass of Python’s `list` built-in. It was meant to represent the actual list of cards that make up a header. That is, it represents an ordered list of cards in the physical order that they appear in the header. It supports the usual list methods for inserting and appending new cards into the list. It also supports `dict`-like keyword access, where `cardlist['KEYWORD']` would return the first card in the list with the given keyword.

A lot of the functionality for manipulating headers was actually buried in the CardList class. The Header class was more of a wrapper around CardList that added a little bit of abstraction. It also implemented a partial dict-like interface, though for Headers a keyword lookup returned the header value associated with that keyword, not the Card object, and almost every method on the Header class was just performing some operations on the underlying CardList.

The problem was that there were certain things a user could *only* do by directly accessing the CardList, such as look up the comments on a card or access cards that have duplicate keywords, such as HISTORY. Another long- standing misfeature was that slicing a Header object actually returned a CardList object, rather than a new Header. For all but the simplest use cases, working with CardList objects was largely unavoidable.

But it was realized that CardList is really an implementation detail not representing any element of a FITS file distinct from the header itself. Users familiar with the FITS format know what a header is, but it is not clear how a “card list” is distinct from that, or why operations go through the Header object, while some have to be performed through the CardList.

So the primary goal of this redesign was to eliminate the `CardList` class altogether, and make it possible for users to perform all header manipulations directly through `Header` objects. It also tried to present headers as similarly as possible to a more familiar data structure — an ordered mapping (or `OrderedDict` in Python) for ease of use by new users less familiar with the FITS format, though there are still many added complexities for dealing with the idiosyncrasies of the FITS format.

Deprecation Warnings

A few older methods on the **`Header`** class have been marked as deprecated, either because they have been renamed to a more PEP 8-compliant name, or because have become redundant due to new features. To check if your code is using any deprecated methods or features, run your code with `python -Wd`. This will output any deprecation warnings to the console.

Two of the most common deprecation warnings related to Headers are:

- `Header.has_key`: this has been deprecated since PyFITS 3.0, just as Python's **`dict.has_key`** is deprecated. To check a key's presence in a mapping object like **`dict`** or **`Header`**, use the `key in d` syntax. This has long been the preference in Python.
- `Header.ascardlist` and `Header.ascard`: these were used to access the `CardList` object underlying a header. They should still work, and return a skeleton CardList implementation that should support most of the old CardList functionality. But try removing as much of this as possible. If direct access to the **`Card`** objects making up a header is necessary, use **`Header.cards`**, which returns an iterator over the cards. More on that below.

New Header Design

The new **`Header`** class is designed to work as a drop-in replacement for a **`dict`** via duck typing. That is, although it is not a subclass of **`dict`**, it implements all of the same methods and interfaces. In particular, it is similar to an **`OrderedDict`** in that the order of insertions is preserved. However, Header also supports many additional features and behaviors specific to the FITS format. It should also be noted that while the old Header implementation also had a dict-like interface, it did not implement the *entire* dict interface as the new Header does.

Although the new Header is used like a dict/mapping in most cases, it also supports a **`list`** interface. The list-like interface is a bit idiosyncratic in that in some contexts the Header acts like a list of values, in others like a list of keywords, and in a few contexts like a list of **`Card`** objects. This may be the most difficult aspect of the new design, but there is a logic to it.

As with the old Header implementation, integer index access is supported: `header[0]` returns the value of the first keyword. However, the **`Header.index()`** method treats the header as though it is a list of keywords and returns the index of a given keyword. For example:

```
>>> header.index('BITPIX')
2
```

Header.count() is similar to **list.count** and also takes a keyword as its argument:

```
>>> header.count('HISTORY')
20
```

A good rule of thumb is that any item access using square brackets `[]` returns *value* in the header, whether using keyword or index lookup. Methods like **index()** and **count()** that deal with the order and quantity of items in the Header generally work on keywords. Finally, methods like **insert()** and **append()** that add new items to the header work on cards.

Aside from the list-like methods, the new Header class works very similarly to the old implementation for most basic use cases and should not present too many surprises. There are differences, however:

- As before, the Header() initializer can take a list of **Card** objects with which to fill the header. However, now any iterable may be used. It is also important to note that *any* Header method that accepts **Card** objects can also accept 2-tuples or 3-tuples in place of Cards. That is, either a `(keyword, value, comment)` tuple or a `(keyword, value)` tuple (comment is assumed blank) may be used anywhere in place of a Card object. This is even preferred, as it involves less typing. For example:

  ```
  >>> from astropy.io import fits
  >>> header = fits.Header([('A', 1), ('B', 2), ('C', 3, 'A
  comment')])
  >>> header
  A       =                    1
  B       =                    2
  C       =                    3 / A comment
  ```

- As demonstrated in the previous example, the `repr()` for a Header (that is, the text that is displayed when entering a Header object in the Python console as an expression), shows the header as it would appear in a FITS file. This inserts newlines after each card so that it is readable regardless of terminal width. It is *not* necessary to use `print header` to view this. Entering `header` displays the header contents as it would appear in the file (sans the END card).
- `len(header)` is now supported (previously it was necessary to do

`len(header.ascard)`). This returns the total number of cards in the header, including blank cards, but excluding the END card.

- FITS supports having duplicate keywords, although they are generally in error except for commentary keywords like COMMENT and HISTORY. PyFITS now supports reading, updating, and deleting duplicate keywords; instead of using the keyword by itself, use a `(keyword, index)` tuple. For example, `('HISTORY', 0)` represents the first HISTORY card, `('HISTORY', 1)` represents the second HISTORY card, and so on. In fact, when a keyword is used by itself, it is shorthand for `(keyword, 0)`. It is now possible to delete an accidental duplicate like so:

  ```
  >>> del header[('NAXIS', 1)]
  ```

  This will remove an accidental duplicate NAXIS card from the header.

- Even if there are duplicate keywords, keyword lookups like `header['NAXIS']` will always return the value associated with the first copy of that keyword, with one exception: commentary keywords like COMMENT and HISTORY are expected to have duplicates. So `header['HISTORY']`, for example, returns the whole sequence of HISTORY values in the correct order. This list of values can be sliced arbitrarily. For example, to view the last three history entries in a header:

  ```
  >>> hdulist[0].header['HISTORY'][-3:]
    reference table oref$laf13367o_pct.fits
    reference table oref$laf13369o_apt.fits
  Heliocentric correction = 16.225 km/s
  ```

- Subscript assignment can now be used to add new keywords to the header. Just as with a normal **dict**, `header['NAXIS'] = 1` will either update the NAXIS keyword if it already exists, or add a new NAXIS keyword with a value of `1` if it does not exist. In the old interface this would return a **KeyError** if NAXIS did not exist, and the only way to add a new keyword was through the update() method.

  By default, new keywords added in this manner are added to the end of the header, with a few FITS-specific exceptions:

  - If the header contains extra blank cards at the end, new keywords are added before the blanks.
  - If the header ends with a list of commentary cards — for example, a sequence of HISTORY cards — those are kept at the end, and new keywords are inserted before the commentary cards.
  - If the keyword is a commentary keyword like COMMENT or HISTORY (or

an empty string for blank keywords), a *new* commentary keyword is always added and appended to the last commentary keyword of the same type. For example, HISTORY keywords are always placed after the last history keyword:

```
>>> header = fits.Header()
>>> header['COMMENT'] = 'Comment 1'
>>> header['HISTORY'] = 'History 1'
>>> header['COMMENT'] = 'Comment 2'
>>> header['HISTORY'] = 'History 2'
>>> header
COMMENT Comment 1
COMMENT Comment 2
HISTORY History 1
HISTORY History 2
```

These behaviors represent a sensible default behavior for keyword assignment, and the same behavior as `update()` in the old Header implementation. The default behaviors may still be bypassed through the use of other assignment methods like the `Header.set()` and `Header.append()` methods described later.

- It is now also possible to assign a value and a comment to a keyword simultaneously using a tuple:

```
>>> header['NAXIS'] = (2, 'Number of axis')
```

This will update the value and comment of an existing keyword, or add a new keyword with the given value and comment.

- There is a new `Header.comments` attribute which lists all of the comments associated with keywords in the header (not to be confused with COMMENT cards). This allows viewing and updating the comments on specific cards:

```
>>> header.comments['NAXIS']
Number of axis
>>> header.comments['NAXIS'] = 'Number of axes'
>>> header.comments['NAXIS']
Number of axes
```

- When deleting a keyword from a header, do not assume that the keyword already exists. In the old Header implementation, this action would silently do nothing. For backwards-compatibility, it is still okay to delete a nonexistent keyword, but a warning will be raised. In the future this *will* be changed so that trying to delete a nonexistent keyword raises a `KeyError`. This is for consistency with the behavior of Python dicts. So unless you know for certain that a keyword exists before deleting it, it is best to do something like:

```
>>> try:
...     del header['BITPIX']
... except KeyError:
...     pass
```

Or if you prefer to look before you leap:

```
>>> if 'BITPIX' in header:
...     del header['BITPIX']
```

- `del header` now supports slices. For example, to delete the last three keywords from a header:

  ```
  >>> del header[-3:]
  ```

- Two headers can now be compared for equality — previously no two Header objects were the same. Now they compare as equal if they contain the exact same content. That is, this requires strict equality.
- Two headers can now be added with the '+' operator, which returns a copy of the left header extended by the right header with **`extend()`**. Assignment addition is also possible.
- The Header.update() method used commonly with the old Header API has been renamed to **`Header.set()`**. The primary reason for this change is very simple: Header implements the **`dict`** interface, which already has a method called update(), but that behaves differently from the old Header.update().

  The details of the new update() can be read in the API docs, but it is very similar to **`dict.update`**. It also supports backwards compatibility with the old update() by analysis of the arguments passed to it, so existing code will not break immediately. However, this *will* cause a deprecation warning to be output if they are enabled. It is best, for starters, to replace all update() calls with set(). Recall, also, that direct assignment is now possible for adding new keywords to a header. So by and large the only reason to prefer using **`Header.set()`** is its capability of inserting or moving a keyword to a specific location using the `before` or `after` arguments.
- Slicing a Header with a slice index returns a new Header containing only those cards contained in the slice. As mentioned earlier, it used to be that slicing a Header returned a card list — something of a misfeature. In general, objects that support slicing ought to return an object of the same type when

you slice them.

Likewise, wildcard keywords used to return a CardList object — now they return a new Header similarly to a slice. For example:

```
>>> header['NAXIS*']
```

returns a new header containing only the NAXIS and NAXISn cards from the original header.

Transition Tips

The above may seem like a lot, but the majority of existing code using PyFITS to manipulate headers should not need to be updated, at least not immediately. The most common operations still work the same.

As mentioned above, it would be helpful to run your code with `python -Wd` to enable deprecation warnings — that should be a good idea of where to look to update your code.

If your code needs to be able to support older versions of PyFITS simultaneously with PyFITS 3.1, things are slightly trickier, but not by much — the deprecated interfaces will not be removed for several more versions because of this.

- The first change worth making, which is supported by any PyFITS version in the last several years, is to remove any use of `Header.has_key` and replace it with `keyword in header` syntax. It is worth making this change for any dict as well, since **`dict.has_key`** is deprecated. Running the following regular expression over your code may help with most (but not all) cases:

  ```
  s/([^ ]+)\.has_key\(([^)]+)\)/\2 in \1/
  ```

- If possible, replace any calls to Header.update() with Header.set() (though do not bother with this if you need to support older PyFITS versions). Also, if you have any calls to Header.update() that can be replaced with simple subscript assignments (e.g., `header['NAXIS'] = (2, 'Number of axes')`) do that too, if possible.
- Find any code that uses `header.ascard` or `header.ascardlist()`. First ascertain whether that code really needs to work directly on Card objects. If that is definitely the case, go ahead and replace those with `header.cards` — that should work without too much fuss. If you do need

to support older versions, you may keep using `header.ascard` for now.

- In the off chance that you have any code that slices a header, it is best to take the result of that and create a new Header object from it. For example:

  ```
  >>> new_header = fits.Header(old_header[2:])
  ```

  This avoids the problem that in PyFITS <= 3.0 slicing a Header returns a CardList by using the result to initialize a new Header object. This will work in both cases (in PyFITS 3.1, initializing a Header with an existing Header just copies it, à la **`list`**).

- As mentioned earlier, locate any code that deletes keywords with `del` and make sure they either look before they leap (`if keyword in header:`) or ask forgiveness (`try/except KeyError:`).

Other Gotchas

- As mentioned above, it is not necessary to enter `print header` to display a header in an interactive Python prompt. Entering `>>> header` by itself is sufficient. Using `print` usually will *not* display the header readably, because it does not include line breaks between the header cards. The reason is that Python has two types of string representations. One is returned when a user calls `str(header)`, which happens automatically when you `print` a variable. In the case of the Header class this actually returns the string value of the header as it is written literally in the FITS file, which includes no line breaks.

  The other type of string representation happens when one calls `repr(header)`. The **`repr`** of an object is meant to be a useful string "representation" of the object; in this case the contents of the header but with line breaks between the cards and with the END card and trailing padding stripped off. This happens automatically when a user enters a variable at the Python prompt by itself without a `print` call.

- The current version of the FITS Standard (3.0) states in section 4.2.1 that trailing spaces in string values in headers are not significant and should be ignored. PyFITS < 3.1 *did* treat trailing spaces as significant. For example, if a header contained:

  > KEYWORD1= 'Value '

  then `header['KEYWORD1']` would return the string `'Value     '` exactly, with the trailing spaces intact. The new Header interface fixes this by automatically stripping trailing spaces, so that `header['KEYWORD1']` would return just `'Value'`.

There is, however, one convention used by the IRAF CCD mosaic task for representing its TNX World Coordinate System and ZPX World Coordinate System nonstandard WCS that uses a series of keywords in the form `WATj_nnn`, which store a text description of coefficients for a nonlinear distortion projection. It uses its own microformat for listing the coefficients as a string, but the string is long, and thus broken up into several of these `WATj_nnn` keywords. Correct recombination of these keywords requires treating all whitespace literally. This convention either overlooked or predated the prescribed treatment of whitespace in the FITS standard.

To get around this issue, a global variable `fits.STRIP_HEADER_WHITESPACE` was introduced. Temporarily setting `fits.STRIP_HEADER_WHITESPACE.set(False)` before reading keywords affected by this issue will return their values with all trailing whitespace intact.

A future version of PyFITS may be able to detect use of conventions like this contextually and behave according to the convention, but in most cases the default behavior of PyFITS is to behave according to the FITS Standard.

astropy.io.fits History

Prior to its inclusion in Astropy, the **`astropy.io.fits`** package was a stand-alone package called PyFITS. PyFITS is no longer actively maintained, and its development is now solely in Astropy. This page documents the release history of PyFITS prior to its merge into Astropy.

PyFITS Changelog

3.4.0 (2016-01-29)

This is the last released version of PyFITS as a standalone package.

3.3.0 (2014-07-17)

New Features

- Added new verification options `fix+ignore`, `fix+warn`, `fix+exception`, `silentfix+ignore`, `silentfix+warn`, and `silentfix+exception` which give more control over how to report fixable errors as opposed to unfixable errors. See the “Verification” section in the PyFITS documentation for more details.

API Changes

- The `pyfits.new_table` function is now fully deprecated (though will not be removed for a long time, considering how widely it is used).

  Instead please use the more explicit `pyfits.BinTableHDU.from_columns` to create a new binary table

HDU, and the similar `pyfits.TableHDU.from_columns` to create a new ASCII table. These otherwise accept the same arguments as `pyfits.new_table` which is now just a wrapper for these.

- The `.fromstring` classmethod of each HDU type has been simplified such that, true to its namesake, it only initializes an HDU from a string containing its header *and* data. (spacetelescope/PyFITS#64)
- Fixed an issue where header wildcard matching (for example `header['DATE*']`) can be used to match *any* characters that might appear in a keyword. Previously this only matched keywords containing characters in the set `[0-9A-Za-z_]`. Now this can also match a hyphen `-` and any other characters, as some conventions like `HIERARCH` and record-valued keyword cards allow a wider range of valid characters than standard FITS keywords.
- This will be the *last* release to support the following APIs that have been marked deprecated since PyFITS v3.1:
    - The `CardList` class, which was part of the old header implementation.
    - The `Card.key` attribute. Use `Card.keyword` instead.
    - The `Card.cardimage` and `Card.ascardimage` attributes. Use simply `Card.image` or `str(card)` instead.
    - The `create_card` factory function. Simply use the normal `Card` constructor instead.
    - The `create_card_from_string` factory function. Use `Card.fromstring` instead.
    - The `upper_key` function. Use `Card.normalize_keyword` method instead (this is not unlikely to be used outside of PyFITS itself, but it was technically public API).
    - The usage of `Header.update` with `Header.update(keyword, value, comment)` arguments. `Header.update` should only be used analogously to `dict.update`. Use `Header.set` instead.
    - The `Header.ascard` attribute. Use `Header.cards` instead for a list of all the `Card` objects in the header.
    - The `Header.rename_key` method. Use `Header.rename_keyword` instead.
    - The `Header.get_history` method. Use `header['HISTORY']` instead (normal keyword lookup).
    - The `Header.get_comment` method. Use `header['COMMENT']` instead.
    - The `Header.toTxtFile` method. Use `header.totextfile` instead.
    - The `Header.fromTxtFile` method. Use `Header.fromtextfile`

instead.

- The `pyfits.tdump` and `tcreate` functions. Use `pyfits.tabledump` and `pyfits.tableload` respectively.
- The `BinTableHDU.tdump` and `tcreate` methods. Use `BinTableHDU.dump` and `BinTableHDU.load` respectively.
- The `txtfile` argument to the `Header` constructor. Use `Header.fromfile` instead.
- The `startColumn` and `endColumn` arguments to the `FITS_record` constructor. These are unlikely to be used by any user code.

These deprecated interfaces will be removed from the development version of PyFITS following the v3.3 release (they will still be available in any v3.3.x bugfix releases, however).

Other Changes and Additions

- PyFITS has switched to a unified code base which supports Python 2.5 through 3.4 simultaneously without translation. This *shouldn't* have any significant performance impacts, but please report if anything seems noticeably slower. As a reminder, support for Python 2.5 will be ended after PyFITS 3.3.x.
- Warnings for deprecated APIs in PyFITS are now always displayed by default. This is in line with a similar change made recently to Astropy: https://github.com/astropy/astropy/pull/1871 To disable PyFITS deprecation warnings in scripts one may call `pyfits.ignore_deprecation_warnings()` after importing PyFITS.
- `Card` objects have a new `is_blank` attribute which returns `True` if the card represents a blank card (no keyword, value, or comment) and `False` otherwise.

Bug Fixes

- Fixed a regression where it was not possible to save an empty “compressed” image to a file (in this case there is nothing to compress, hence the quotes, but trying to do so caused a crash). (spacetelescope/PyFITS#69)
- Fixed a regression that may have been introduced in v3.2.1 with writing compressed image HDUs, particularly compressed images using a non-empty GZIP_COMPRESSED_DATA column. (spacetelescope/#71)

3.2.4 (2014-06-02)

- Fixed a regression where multiple consecutive calls of the `writeto` method on the same HDU but to different files could lead to corrupt data or crashes on the subsequent calls after the first. (spacetelescope/PyFITS#40)

3.2.3 (2014-05-14)

- Nominal support for Python 3.4.
- Fixed a bug with using the `tabledump` and `tableload` functions with tables containing array columns (columns in which each element is an array instead of a single scalar value). (spacetelescope/PyFITS#22)
- Fixed an issue where PyFITS allowed newline characters in header values and comments. (spacetelescope/PyFITS#51)
- Fixed pickling of `FITS_rec` (table data) objects. (spacetelescope/PyFITS#53)
- Improved behavior when writing large compressed images on OSX by removing an unnecessary check for platform architecture. (spacetelescope/PyFITS#57)
- Allow reading FITS files from file-like objects that do not have a `.closed` attribute (and as such may not even have an "open" vs. "closed" concept). (spacetelescope/PyFITS#56)
- Fixed duplicate insertion of commentary keywords on compressed image headers. (spacetelescope/PyFITS#58)
- Fixed minor issue with comparison of header commentary card values. (spacetelescope/PyFITS#59)

3.1.6 (2014-05-14)

- Nominal support for Python 3.4.
- Fixed a bug with using the `tabledump` and `tableload` functions with tables containing array columns (columns in which each element is an array instead of a single scalar value). (Backported from 3.2.3)
- Fixed an issue where PyFITS allowed newline characters in header values and comments. (Backported from 3.2.3)
- Fixed pickling of `FITS_rec` (table data) objects. (Backported from 3.2.3)
- Improved behavior when writing large compressed images on OSX by removing an unnecessary check for platform architecture. (Backported from 3.2.3)
- Allow reading FITS files from file-like objects that do not have a `.closed` attribute (and as such may not even have an "open" vs. "closed" concept). (Backported from 3.2.3)
- Fixed minor issue with comparison of header commentary card values.

(Backported from 3.2.3)

3.2.2 (2014-03-25)

- Fixed a regression on deletion of record-valued keyword cards using the Header wildcard syntax. This was intended to be fixed before the v3.2.1 release.

3.1.5 (2014-03-25)

- Fixed a regression on deletion of record-valued keyword cards using the Header wildcard syntax. This was intended to be fixed before the v3.1.4 release.

3.2.1 (2014-03-04)

- Nominal support for the upcoming Python 3.4.
- Added missing features from the `Header.insert()` method that were intended for inclusion in the original 3.1 release: In addition to accepting an integer index as the first argument, it also supports supplying a keyword name as the first argument for insertion relative to a specific keyword. It also now supports an optional `after` argument. If `after=True` the insertion is made below the insertion point instead of above it. (spacetelescope/PyFITS#12)
- Fixed support for broadcasting of values assigned to table columns. (spacetelescope/PyFITS#48)
- A grab bag of minor performance improvements in headers. (spacetelescope/PyFITS#46)
- Fix an unrelated error that occurred when instantiating a `ColDefs` object with invalid input.
- Fixed an issue where opening an image containing pseudo-unsigned integers and immediately writing it to a new file using the `writeto` method would drop the scale factors that identified the data as unsigned.
- Fixed a bug where writing a file with `checksum=True` did not add the checksum on new files. (spacetelescope/PyFITS#8)
- Fixed an issue where validating an HDU's checksums removed the checksum from that HDU's header entirely (even if it was valid.)

- Fixed checksums on compressed images, so that the `ZHECKSUM` and `ZDATASUM` contain a checksum of the original image HDU, while `CHECKSUM` and `DATASUM` contain checksums of the compressed image HDU. This feature was supposed to be supported in 3.2, but the support was buggy.
- Fixed an issue where the size of the heap was sometimes not computed properly when writing an existing table containing variable-length array columns to a new FITS file. This could result in corruption in the new FITS file. (spacetelescope/PyFITS#47)
- Fixed issue with updates to the header of `CompImageHDU` objects not being preserved on save. (spacetelescope/PyFITS#23)
- Fixed a bug where a boolean value of `True` in a header could not be replaced with the integer 1, and likewise for `False` and 0 and vice versa.
- Fixed an issue similar to the above one but for numeric values–now replacing a header value with an equivalent numeric value will up/downcast that value. For example replacing ‘0’ with ‘0.0’ will write ‘0.0’ to the header so that it is returned as a floating point value. Likewise a float can be downcast to an integer. (spacetelescope/PyFITS#49)
- A handful of Python 3 compatibility fixes, especially for compatibility with the upcoming Python 3.4.
- Fixed unrelated crash when a header contains an invalid END card (for example “END = “). This resulted in a cryptic traceback. Now headers like this will detect “clearly intended” END cards and produce a warning about their invalidity and fix them. (#217)
- Allowed a sequence of `Column` objects to be passed in as the main argument to `FITS_rec.from_columns` as the documentation suggests should be possible.
- Fixed a display formatting issue with fitsdiff where sometimes it did not show the difference between two floating point numbers if they were the same up to some low number of digits. (spacetelescope/PyFITS#21)
- Fixed an issue where Python 2 sometimes allowed non-ASCII strings to be assigned as header values if they were assigned as old-style `str` objects and not `unicode` objects. (spacetelescope/PyFITS#37)

3.1.4 (2014-03-04)

- Added missing features from the `Header.insert()` method that were intended for inclusion in the original 3.1 release: In addition to accepting an integer index as the first argument, it also supports supplying a keyword name as the first argument for insertion relative to a specific keyword. It also

now supports an optional `after` argument. If `after=True` the insertion is made below the insertion point instead of above it. (Backported from 3.2.1)
- A grab bag of minor performance improvements in headers. (Backported from 3.2.1)
- Fixed an issue where opening an image containing pseudo-unsigned integers and immediately writing it to a new file using the `writeto` method would drop the scale factors that identified the data as unsigned. (Backported from 3.2.1)
- Fixed a bug where writing a file with `checksum=True` did not add the checksum on new files. (Backported from 3.2.1)
- Fixed an issue where validating an HDU's checksums removed the checksum from that HDU's header entirely (even if it was valid.) (Backported from 3.2.1)
- Fixed an issue where the size of the heap was sometimes not computed properly when writing an existing table containing variable-length array columns to a new FITS file. This could result in corruption in the new FITS file. (Backported from 3.2.1)
- Fixed a bug where a boolean value of `True` in a header could not be replaced with the integer 1, and likewise for `False` and 0 and vice versa. (Backported from 3.2.1)
- Fixed an issue similar to the above one but for numeric values–now replacing a header value with an equivalent numeric value will up/downcast that value. For example replacing '0' with '0.0' will write '0.0' to the header so that it is returned as a floating point value. Likewise a float can be downcast to an integer. (Backported from 3.2.1)
- Fixed unrelated crash when a header contains an invalid END card (for example "END = "). This resulted in a cryptic traceback. Now headers like this will detect "clearly intended" END cards and produce a warning about their invalidity and fix them. (Backported from 3.2.1)
- Fixed a display formatting issue with fitsdiff where sometimes it did not show the difference between two floating point numbers if they were the same up to some low number of digits. (Backported from 3.2.1)
- Fixed an issue where Python 2 sometimes allowed non-ASCII strings to be assigned as header values if they were assigned as old-style `str` objects and not `unicode` objects. (Backported from 3.2.1)

3.0.13 (2014-03-04)

- Fixed a bug where writing a file with `checksum=True` did not add the checksum on new files. (Backported from 3.2.1)
- Fixed an issue where validating an HDU's checksums removed the

checksum from that HDU’s header entirely (even if it was valid.) (Backported from 3.2.1)

3.2 (2013-11-26)

Highlights

- Rewrote CFITSIO-based backend for handling tile compression of FITS files. It now uses a standard CFITSIO instead of heavily modified pieces of CFITSIO as before. PyFITS ships with its own copy of CFITSIO v3.35 which supports the latest version of the Tiled Image Convention (v2.3), but system packagers may choose instead to strip this out in favor of a system-installed version of CFITSIO. Earlier versions may work, but nothing earlier than 3.28 has been tested yet. (#169)
- Added support for reading and writing tables using the Q format for columns. The Q format is identical to the P format (variable-length arrays) except that it uses 64-bit integers for the data descriptors, allowing more than 4 GB of variable-length array data in a single table. (#160)
- Added initial support for table columns containing pseudo-unsigned integers. This is currently enabled by using the `uint=True` option when opening files; any table columns with the correct BZERO value will be interpreted and returned as arrays of unsigned integers.
- Some refactoring of the table and `FITS_rec` modules in order to better separate the details of the FITS binary and ASCII table data structures from the HDU data structures that encapsulate them. Most of these changes should not be apparent to users (but see API Changes below).

API Changes

- Assigning to values in `ColDefs.names`, `ColDefs.formats`, `ColDefs.nulls` and other attributes of `ColDefs` instances that return lists of column properties is no longer supported. Assigning to those lists will no longer update the corresponding columns. Instead, please just modify the `Column` instances directly (`Column.name`, `Column.null`, etc.)
- The `pyfits.new_table` function is marked “pending deprecation”. This does not mean it will be removed outright or that its functionality has changed. It will likely be replaced in the future for a function with similar, if not subtly different functionality. A better, if not slightly more verbose approach is to use `pyfits.FITS_rec.from_columns` to create a new `FITS_rec` table–this has the same interface as `pyfits.new_table`. The difference is that it returns a plan `FITS_rec` array, and not an HDU

instance. This `FITS_rec` object can then be used as the data argument in the constructors for `BinTableHDU` (for binary tables) or `TableHDU` (for ASCII tables). This is analogous to creating an `ImageHDU` by passing in an image array. `pyfits.FITS_rec.from_columns` is just a simpler way of creating a FITS-compatible recarray from a FITS column specification.

- The `updateHeader`, `updateHeaderData`, and `updateCompressedData` methods of the `CompDataHDU` class are pending deprecation and moved to internal methods. The operation of these methods depended too much on internal state to be used safely by users; instead they are invoked automatically in the appropriate places when reading/writing compressed image HDUs.
- The `CompDataHDU.compData` attribute is pending deprecation in favor of the clearer and more PEP-8 compatible `CompDataHDU.compressed_data`.
- The constructor for `CompDataHDU` has been changed to accept new keyword arguments. The new keyword arguments are essentially the same, but are in underscore_separated format rather than camelCase format. The old arguments are still pending deprecation.
- The internal attributes of HDU classes `_hdrLoc`, `_datLoc`, and `_datSpan` have been replaced with `_header_offset`, `_data_offset`, and `_data_size` respectively. The old attribute names are still pending deprecation. This should only be of interest to advanced users who have created their own HDU subclasses.
- The following previously deprecated functions and methods have been removed entirely: `createCard`, `createCardFromString`, `upperKey`, `ColDefs.data`, `setExtensionNameCaseSensitive`, `_File.getfile`, `_TableBaseHDU.get_coldefs`, `Header.has_key`, `Header.ascardlist`.

  If you run your code with a previous version of PyFITS (>= 3.0, < 3.2) with the `python -Wd` argument, warnings for all deprecated interfaces still in use will be displayed.
- Interfaces that were pending deprecation are now fully deprecated. These include: `create_card`, `create_card_from_string`, `upper_key`, `Header.get_history`, and `Header.get_comment`.
- The `.name` attribute on HDUs is now directly tied to the HDU's header, so that if `.header['EXTNAME']` changes so does `.name` and vice-versa.
- The `pyfits.file.PYTHON_MODES` constant dict was renamed to `pyfits.file.PYFITS_MODES` which better reflects its purpose. This is

rarely used by client code, however. Support for the old name will be removed by PyFITS 3.4.

Other Changes and Additions

- The new compression code also adds support for the ZQUANTIZ and ZDITHER0 keywords added in more recent versions of this FITS Tile Compression spec. This includes support for lossless compression with GZIP. (#198) By default no dithering is used, but the `SUBTRACTIVE_DITHER_1` and `SUBTRACTIVE_DITHER_2` methods can be enabled by passing the correct constants to the `quantize_method` argument to the `CompImageHDU` constructor. A seed can be manually specified, or automatically generated using either the system clock or checksum-based methods via the `dither_seed` argument. See the documentation for `CompImageHDU` for more details. (#198) (spacetelescope/PYFITS#32)
- Images compressed with the Tile Compression standard can now be larger than 4 GB through support of the Q format. (#159)
- All HDUs now have a `.ver` `.level` attribute that returns the value of the EXTVAL and EXTLEVEL keywords from that HDU's header, if the exist. This was added for consistency with the `.name` attribute which returns the EXTNAME value from the header.
- Then `Column` and `ColDefs` classes have new `.dtype` attributes which give the Numpy dtype for the column data in the first case, and the full Numpy compound dtype for each table row in the latter case.
- There was an issue where new tables created defaulted the values in all string columns to '0.0'. Now string columns are filled with empty strings by default–this seems a less surprising default, but it may cause differences with tables created with older versions of PyFITS.
- Improved round-tripping and preservation of manually assigned column attributes (`TNULLn` , `TSCALn` , etc.) in table HDU headers. (astropy/astropy#996)

Bug Fixes

- Binary tables containing compressed images may, optionally, contain other columns unrelated to the tile compression convention. Although this is an uncommon use case, it is permitted by the standard. (#159)
- Reworked some of the file I/O routines to allow simpler, more consistent mapping between OS-level file modes ('rb', 'wb', 'ab', etc.) and the more "PyFITS-specific" modes used by PyFITS like "readonly" and "update". That is, if reading a FITS file from an open file object, it doesn't matter as much what "mode" it was opened in so long as it has the right capabilities (read/write/etc.) Also works around bugs in the Python io module in 2.6+ with

regard to file modes. (spacetelescope/PyFITS#33)
- Fixed an obscure issue that can occur on systems that don't have flush to memory-mapped files implemented (namely GNU Hurd). (astropy/astropy#968)

3.1.3 (2013-11-26)

- Disallowed assigning NaN and Inf floating point values as header values, since the FITS standard does not define a way to represent them in. Because this is undefined, the previous behavior did not make sense and produced invalid FITS files. (spacetelescope/PyFITS#11)
- Added a workaround for a bug in 64-bit OSX that could cause truncation when writing files greater than 2^32 bytes in size. (spacetelescope/PyFITS#28)
- Fixed a long-standing issue where writing binary tables did not correctly write the TFORMn keywords for variable-length array columns (they omitted the max array length parameter of the format). This was thought fixed in v3.1.2, but it was only fixed there for compressed image HDUs and not for binary tables in general.
- Fixed an obscure issue that can occur on systems that don't have flush to memory-mapped files implemented (namely GNU Hurd). (Backported from 3.2)

3.0.12 (2013-11-26)

- Disallowed assigning NaN and Inf floating point values as header values, since the FITS standard does not define a way to represent them in. Because this is undefined, the previous behavior did not make sense and produced invalid FITS files. (Backported from 3.1.3)
- Added a workaround for a bug in 64-bit OSX that could cause truncation when writing files greater than 2^32 bytes in size. (Backported from 3.1.3)
- Fixed a long-standing issue where writing binary tables did not correctly write the TFORMn keywords for variable-length array columns (they omitted the max array length parameter of the format). This was thought fixed in v3.1.2, but it was only fixed there for compressed image HDUs and not for binary tables in general. (Backported from 3.1.3)
- Fixed an obscure issue that can occur on systems that don't have flush to memory-mapped files implemented (namely GNU Hurd). (Backported from 3.2)

3.1.3 (unreleased)

- Disallowed assigning NaN and Inf floating point values as header values, since the FITS standard does not define a way to represent them in. Because this is undefined, the previous behavior did not make sense and produced invalid FITS files. (spacetelescope/PyFITS#11)

3.0.12 (unreleased)

- Disallowed assigning NaN and Inf floating point values as header values, since the FITS standard does not define a way to represent them in. Because this is undefined, the previous behavior did not make sense and produced invalid FITS files. (Backported from 3.1.3)
- Added a workaround for a bug in 64-bit OSX that could cause truncation when writing files greater than 2^32 bytes in size. (Backported from 3.1.3)

3.1.2 (2013-04-22)

- When an error occurs opening a file in fitsdiff the exception message will now at least mention which file had the error. (#168)
- Fixed support for opening gzipped FITS files by filename in a writeable mode (PyFITS has supported writing to gzip files for some time now, but only enabled it when GzipFile objects were passed to `pyfits.open()` due to some legacy code preventing full gzip support. (#195)
- Added a more helpful error message in the case of malformatted FITS files that contain non-float NULL values in an ASCII table but are missing the required TNULLn keywords in the header. (#197)
- Fixed an (apparently long-standing) issue where writing compressed images did not correctly write the TFORMn keywords for variable-length array columns (they omitted the max array length parameter of the format). (#199)
- Slightly refactored how tables containing variable-length array columns are handled to add two improvements: Fixes an issue where accessing the data after a call to the `pyfits.getdata` convenience function caused an exception, and allows the VLA data to be read from an existing mmap of the FITS file. (#200)
- Fixed a bug that could occur when opening a table containing multi-dimensional columns (i.e. via the TDIMn keyword) and then writing it out to a new file. (#201)
- Added use of the console_scripts entry point to install the fitsdiff and

fitscheck scripts, which if nothing else provides better Windows support. The generated scripts now override the ones explicitly defined in the scripts/ directory (which were just trivial stubs to begin with). (#202)
- Fixed a bug on Python 3 where attempting to open a non-existent file on Python 3 caused a seemingly unrelated traceback. (#203)
- Fixed a bug in fitsdiff that reported two header keywords containing NaN as value as different. (#204)
- Fixed an issue in the tests that caused some tests to fail if pyfits is installed with read-only permissions. (#208)
- Fixed a bug where instantiating a `BinTableHDU` from a numpy array containing boolean fields converted all the values to `False`. (#215)
- Fixed an issue where passing an array of integers into the constructor of `Column()` when the column type is floats of the same byte width caused the column array to become garbled. (#218)
- Fixed inconsistent behavior in creating CONTINUE cards from byte strings versus Unicode strings in Python 2–CONTINUE cards can now be created properly from Unicode strings (so long as they are convertible to ASCII). (spacetelescope/PyFITS#1)
- Fixed a couple cases where creating a new table using TDIMn in some of the columns could caused a crash. (spacetelescope/PyFITS#3)
- Fixed a bug in parsing HIERARCH keywords that do not have a space after the first equals sign (before the value). (spacetelescope/PyFITS#5)
- Prevented extra leading whitespace on HIERARCH keywords from being treated as part of the keyword. (spacetelescope/PyFITS#6)
- Fixed a bug where HIERARCH keywords containing lower-case letters was mistakenly marked as invalid during header validation. (spacetelescope/PyFITS#7)
- Fixed an issue that was ancillary to (spacetelescope/PyFITS#7) where the `Header.index()` method did not work correctly with HIERARCH keywords containing lower-case letters.

3.0.11 (2013-04-17)

- Fixed support for opening gzipped FITS files by filename in a writeable mode (PyFITS has supported writing to gzip files for some time now, but only enabled it when GzipFile objects were passed to `pyfits.open()` due to some legacy code preventing full gzip support. Backported from 3.1.2. (#195)
- Added a more helpful error message in the case of malformatted FITS files that contain non-float NULL values in an ASCII table but are missing the required TNULLn keywords in the header. Backported from 3.1.2. (#197)

- Fixed an (apparently long-standing) issue where writing compressed images did not correctly write the TFORMn keywords for variable-length array columns (they omitted the max array length parameter of the format). Backported from 3.1.2. (#199)
- Slightly refactored how tables containing variable-length array columns are handled to add two improvements: Fixes an issue where accessing the data after a call to the `pyfits.getdata` convenience function caused an exception, and allows the VLA data to be read from an existing mmap of the FITS file. Backported from 3.1.2. (#200)
- Fixed a bug that could occur when opening a table containing multi-dimensional columns (i.e. via the TDIMn keyword) and then writing it out to a new file. Backported from 3.1.2. (#201)
- Fixed a bug on Python 3 where attempting to open a non-existent file on Python 3 caused a seemingly unrelated traceback. Backported from 3.1.2. (#203)
- Fixed a bug in fitsdiff that reported two header keywords containing NaN as value as different. Backported from 3.1.2. (#204)
- Fixed an issue in the tests that caused some tests to fail if pyfits is installed with read-only permissions. Backported from 3.1.2. (#208)
- Fixed a bug where instantiating a `BinTableHDU` from a numpy array containing boolean fields converted all the values to `False`. Backported from 3.1.2. (#215)
- Fixed an issue where passing an array of integers into the constructor of `Column()` when the column type is floats of the same byte width caused the column array to become garbled. Backported from 3.1.2. (#218)
- Fixed a couple cases where creating a new table using TDIMn in some of the columns could caused a crash. Backported from 3.1.2. (spacetelescope/PyFITS#3)

3.1.1 (2013-01-02)

This is a bug fix release for the 3.1.x series.

Bug Fixes

- Improved handling of scaled images and pseudo-unsigned integer images in compressed image HDUs. They now work more transparently like normal image HDUs with support for the `do_not_scale_image_data` and `uint` options, as well as `scale_back` and `save_backup`. The `.scale()` method works better too. (#88)
- Permits non-string values for the EXTNAME keyword when reading in a file, rather than throwing an exception due to the malformatting. Added

verification for the format of the EXTNAME keyword when writing. (#96)

- Added support for EXTNAME and EXTVER in PRIMARY HDUs. That is, if EXTNAME is specified in the header, it will also be reflected in the `.name` attribute and in `pyfits.info()`. These keywords used to be verboten in PRIMARY HDUs, but the latest version of the FITS standard allows them. (#151)
- HCOMPRESS can again be used to compress data cubes (and higher-dimensional arrays) so long as the tile size is effectively 2-dimensional. In fact, PyFITS will automatically use compatible tile sizes even if they're not explicitly specified. (#171)
- Added support for the optional `endcard` parameter in the `Header.fromtextfile()` and `Header.totextfile()` methods. Although `endcard=False` was a reasonable default assumption, there are still text dumps of FITS headers that include the END card, so this should have been more flexible. (#176)
- Fixed a crash when running fitsdiff on two empty (that is, zero row) tables. (#178)
- Fixed an issue where opening files containing random groups HDUs in update mode could cause an unnecessary rewrite of the file even if none of the data is modified. (#179)
- Fixed a bug that could caused a deadlock in the filesystem on OSX if PyFITS is used with Numpy 1.7 in some cases. (#180)
- Fixed a crash when generating diff reports from diffs using the `ignore_comments` options. (#181)
- Fixed some bugs with FITS WCS distortion paper record-valued keyword cards:
  - Cards that looked kind of like RVKCs but were not intended to be were over-permissively treated as such–commentary keywords like COMMENT and HISTORY were particularly affected. (#183)
  - Looking up a card in a header by its standard FITS keyword only should always return the raw value of that card. That way cards containing values that happen to valid RVKCs but were not intended to be will still be treated like normal cards. (#184)
  - Looking up a RVKC in a header with only part of the field-specifier (for example “DP1.AXIS” instead of “DP1.AXIS.1”) was implicitly treated as a wildcard lookup. (#184)
- Fixed a crash when diffing two FITS files where at least one contains a compressed image HDU which was not recognized as an image instead of a table. (#187)
- Fixed bugs in the backwards compatibility layer for the `CardList.index` and `CardList.count` methods. (#190)
- Improved `__repr__` and text file representation of cards with long values

that are split into CONTINUE cards. (#193)
- Fixed a crash when trying to assign a long (> 72 character) value to blank (‘’) keywords. This also changed how blank keywords are represented–there are still exactly 8 spaces before any commentary content can begin; this *may* affect the exact display of header cards that assumed there could be fewer spaces in a blank keyword card before the content begins. However, the current approach is more in line with the requirements of the FITS standard. (#194)

3.0.10 (2013-01-02)

- Improved handling of scaled images and pseudo-unsigned integer images in compressed image HDUs. They now work more transparently like normal image HDUs with support for the `do_not_scale_image_data` and `uint` options, as well as `scale_back` and `save_backup`. The `.scale()` method works better too. Backported from 3.1.1. (#88)
- Permits non-string values for the EXTNAME keyword when reading in a file, rather than throwing an exception due to the malformatting. Added verification for the format of the EXTNAME keyword when writing. Backported from 3.1.1. (#96)
- Added support for EXTNAME and EXTVER in PRIMARY HDUs. That is, if EXTNAME is specified in the header, it will also be reflected in the `.name` attribute and in `pyfits.info()`. These keywords used to be verbotten in PRIMARY HDUs, but the latest version of the FITS standard allows them. Backported from 3.1.1. (#151)
- HCOMPRESS can again be used to compress data cubes (and higher-dimensional arrays) so long as the tile size is effectively 2-dimensional. In fact, PyFITS will not automatically use compatible tile sizes even if they’re not explicitly specified. Backported from 3.1.1. (#171)
- Fixed a bug when writing out files containing zero-width table columns, where the TFIELDS keyword would be updated incorrectly, leaving the table largely unreadable. Backported from 3.1.0. (#174)
- Fixed an issue where opening files containing random groups HDUs in update mode could cause an unnecessary rewrite of the file even if none of the data is modified. Backported from 3.1.1. (#179)
- Fixed a bug that could caused a deadlock in the filesystem on OSX if PyFITS is used with Numpy 1.7 in some cases. Backported from 3.1.1. (#180)

3.1 (2012-08-08)

Highlights

- The `Header` object has been significantly reworked, and `CardList` objects are now deprecated (their functionality folded into the `Header` class). See API Changes below for more details.
- Memory maps are now used by default to access HDU data. See API Changes below for more details.
- Now includes a new version of the `fitsdiff` program for comparing two FITS files, and a new FITS comparison API used by `fitsdiff`. See New Features below.

API Changes

- The `Header` class has been rewritten, and the `CardList` class is deprecated. Most of the basic details of working with FITS headers are unchanged, and will not be noticed by most users. But there are differences in some areas that will be of interest to advanced users, and to application developers. For full details of the changes, see the "Header Interface Transition Guide" section in the PyFITS documentation. See ticket #64 on the PyFITS Trac for further details and background. Some highlights are listed below:
  - The Header class now fully implements the Python dict interface, and can be used interchangeably with a dict, where the keys are header keywords.
  - New keywords can be added to the header using normal keyword assignment (previously it was necessary to use `Header.update` to add new keywords). For example:

    ```
    >>> header['NAXIS'] = 2
    ```

    will update the existing 'FOO' keyword if it already exists, or add a new one if it doesn't exist, just like a dict.
  - It is possible to assign both a value and a comment at the same time using a tuple:

    ```
    >>> header['NAXIS'] = (2, 'Number of axes')
    ```

  - To add/update a new card and ensure it's added in a specific location, use `Header.set()`:

    ```
    >>> header.set('NAXIS', 2, 'Number of axes', after='BITPIX')
    ```

    This works the same as the old `Header.update()`. `Header.update()` still works in the old way too, but is deprecated.

    - Although `Card` objects still exist, it generally is not necessary to work with them directly. `Header.ascardlist()`/`Header.ascard` are deprecated and should not be used. To directly access the `Card` objects in a header, use `Header.cards`.
    - To access card comments, it is still possible to either go through the card itself, or through `Header.comments`. For example:

      ```
      >>> header.cards['NAXIS'].comment
      Number of axes
      >>> header.comments['NAXIS']
      Number of axes
      ```

    - `Card` objects can now be used interchangeably with `(keyword, value, comment)` 3-tuples. They still have `.value` and `.comment` attributes as well. The `.key` attribute has been renamed to `.keyword` for consistency, though `.key` is still supported (but deprecated).
- Memory mapping is now used by default to access HDU data. That is, `pyfits.open()` uses `memmap=True` as the default. This provides better performance in the majority of use cases–there are only some I/O intensive applications where it might not be desirable. Enabling mmap by default also enabled finding and fixing a large number of bugs in PyFITS' handling of memory-mapped data (most of these bug fixes were backported to PyFITS 3.0.5). (#85)
    - A new `pyfits.USE_MEMMAP` global variable was added. Set `pyfits.USE_MEMMAP = False` to change the default memmap setting for opening files. This is especially useful for controlling the behavior in applications where pyfits is deeply embedded.
    - Likewise, a new `PYFITS_USE_MEMMAP` environment variable is supported. Set `PYFITS_USE_MEMMAP = 0` in your environment to change the default behavior.
- The `size()` method on HDU objects is now a `.size` property–this returns the size in bytes of the data portion of the HDU, and in most cases is equivalent to `hdu.data.nbytes` (#83)
- `BinTableHDU.tdump` and `BinTableHDU.tcreate` are deprecated–use `BinTableHDU.dump` and `BinTableHDU.load` instead. The new methods output the table data in a slightly different format from previous versions, which places quotes around each value. This format is compatible with data dumps from previous versions of PyFITS, but not vice-versa due to a parsing bug in older versions.
- Likewise the `pyfits.tdump` and `pyfits.tcreate` convenience

function versions of these methods have been renamed `pyfits.tabledump` and `pyfits.tableload`. The old deprecated, but currently retained for backwards compatibility. (r1125)

- A new global variable `pyfits.EXTENSION_NAME_CASE_SENSITIVE` was added. This serves as a replacement for `pyfits.setExtensionNameCaseSensitive` which is not deprecated and may be removed in a future version. To enable case-sensitivity of extension names (i.e. treat ‘sci’ as distinct from ‘SCI’) set `pyfits.EXTENSION_NAME_CASE_SENSITIVE = True`. The default is `False`. (r1139)
- A new global configuration variable `pyfits.STRIP_HEADER_WHITESPACE` was added. By default, if a string value in a header contains trailing whitespace, that whitespace is automatically removed when the value is read. Now if you set `pyfits.STRIP_HEADER_WHITESPACE = False` all whitespace is preserved. (#146)
- The old `classExtensions` extension mechanism (which was deprecated in PyFITS 3.0) is removed outright. To our knowledge it was no longer used anywhere. (r1309)
- Warning messages from PyFITS issued through the Python warnings API are now output to stderr instead of stdout, as is the default. PyFITS no longer modifies the default behavior of the warnings module with respect to which stream it outputs to. (r1319)
- The `checksum` argument to `pyfits.open()` now accepts a value of ‘remove’, which causes any existing CHECKSUM/DATASUM keywords to be ignored, and removed when the file is saved.

New Features

- Added support for the proposed “FITS” extension HDU type. FITS HDUs contain an entire FITS file embedded in their data section. `FitsHDU` objects work like other HDU types in PyFITS. Their `.data` attribute returns the raw data array. However, they have a special `.hdulist` attribute which processes the data as a FITS file and returns it as an in-memory HDUList object. FitsHDU objects also support a `FitsHDU.fromhdulist()` classmethod which returns a new `FitsHDU` object that embeds the supplied HDUList. (#80)
- Added a new `.is_image` attribute on HDU objects, which is True if the HDU data is an ‘image’ as opposed to a table or something else. Here the meaning of ‘image’ is fairly loose, and mostly just means a Primary or Image extension HDU, or possibly a compressed image HDU (#71)

- Added an `HDUList.fromstring` classmethod which can parse a FITS file already in memory and instantiate and `HDUList` object from it. This could be useful for integrating PyFITS with other libraries that work on FITS file, such as CFITSIO. It may also be useful in streaming applications. The name is a slight misnomer, in that it actually accepts any Python object that implements the buffer interface, which includes `bytes`, `bytearray`, `memoryview`, `numpy.ndarray`, etc. (#90)
- Added a new `pyfits.diff` module which contains facilities for comparing FITS files. One can use the `pyfits.diff.FITSDiff` class to compare two FITS files in their entirety. There is also a `pyfits.diff.HeaderDiff` class for just comparing two FITS headers, and other similar interfaces. See the PyFITS Documentation for more details on this interface. The `pyfits.diff` module powers the new `fitsdiff` program installed with PyFITS. After installing PyFITS, run `fitsdiff --help` for usage details.
- `pyfits.open()` now accepts a `scale_back` argument. If set to `True`, this automatically scales the data using the original BZERO and BSCALE parameters the file had when it was first opened, if any, as well as the original BITPIX. For example, if the original BITPIX were 16, this would be equivalent to calling `hdu.scale('int16', 'old')` just before calling `flush()` or `close()` on the file. This option applies to all HDUs in the file. (#120)
- `pyfits.open()` now accepts a `save_backup` argument. If set to `True`, this automatically saves a backup of the original file before flushing any changes to it (this of course only applies to update and append mode). This may be especially useful when working with scaled image data. (#121)

Changes in Behavior

- Warnings from PyFITS are not output to stderr by default, instead of stdout as it has been for some time. This is contrary to most users' expectations and makes it more difficult for them to separate output from PyFITS from the desired output for their scripts. (r1319)

Bug Fixes

- Fixed `pyfits.tcreate()` (now `pyfits.tableload()`) to be more robust when encountering blank lines in a column definition file (#14)
- Fixed a fairly rare crash that could occur in the handling of CONTINUE cards when using Numpy 1.4 or lower (though 1.4 is the oldest version supported by PyFITS). (r1330)
- Fixed `_BaseHDU.fromstring` to actually correctly instantiate an HDU object from a string/buffer containing the header and data of that HDU. This allowed for the implementation of `HDUList.fromstring` described above.

(#90)
- Fixed a rare corner case where, in some use cases, (mildly, recoverable) malformatted float values in headers were not properly returned as floats. (#137)
- Fixed a corollary to the previous bug where float values with a leading zero before the decimal point had the leading zero unnecessarily removed when saving changes to the file (eg. “0.001” would be written back as “.001” even if no changes were otherwise made to the file). (#137)
- When opening a file containing CHECKSUM and/or DATASUM keywords in update mode, the CHECKSUM/DATASUM are updated and preserved even if the file was opened with checksum=False. This change in behavior prevents checksums from being unintentionally removed. (#148)
- Fixed a bug where `ImageHDU.scale(option='old')` wasn’t working at all–it was not restoring the image to its original BSCALE and BZERO values. (#162)
- Fixed a bug when writing out files containing zero-width table columns, where the TFIELDS keyword would be updated incorrectly, leaving the table largely unreadable. This fix will be backported to the 3.0.x series in version 3.0.10. (#174)

3.0.9 (2012-08-06)

This is a bug fix release for the 3.0.x series.

Bug Fixes

- Fixed `Header.values()`/`Header.itervalues()` and `Header.items()`/`Header.iteritems()` to correctly return the different values for duplicate keywords (particularly commentary keywords like HISTORY and COMMENT). This makes the old Header implementation slightly more compatible with the new implementation in PyFITS 3.1. (#127)

  Note

  This fix did not change the existing behavior from earlier PyFITS versions where `Header.keys()` returns all keywords in the header with duplicates removed. PyFITS 3.1 changes that behavior, so that `Header.keys()` includes duplicates.
- Fixed a bug where `ImageHDU.scale(option='old')` wasn’t working at all–it was not restoring the image to its original BSCALE and BZERO values. (#162)
- Fixed a bug where opening a file containing compressed image HDUs in

'update' mode and then immediately closing it without making any changes caused the file to be rewritten unnecessarily. (#167)

- Fixed two memory leaks that could occur when writing compressed image data, or in some cases when opening files containing compressed image HDUs in 'update' mode. (#168)

3.0.8 (2012-06-04)

Changes in Behavior

- Prior to this release, image data sections did not work with scaled data–that is, images with non-trivial BSCALE and/or BZERO values. Previously, in order to read such images in sections, it was necessary to manually apply the BSCALE+BZERO to each section. It's worth noting that sections *did* support pseudo-unsigned ints (flakily). This change just extends that support for general BSCALE+BZERO values.

Bug Fixes

- Fixed a bug that prevented updates to values in boolean table columns from being saved. This turned out to be a symptom of a deeper problem that could prevent other table updates from being saved as well. (#139)
- Fixed a corner case in which a keyword comment ending with the string "END" could, in some circumstances, cause headers (and the rest of the file after that point) to be misread. (#142)
- Fixed support for scaled image data and pseudo-unsigned ints in image data sections (`hdu.section`). Previously this was not supported at all. At some point support was supposedly added, but it was buggy and incomplete. Now the feature seems to work much better. (#143)
- Fixed the documentation to point out that image data sections *do* support non-contiguous slices (and have for a long time). The documentation was never updated to reflect this, and misinformed users that only contiguous slices were supported, leading to some confusion. (#144)
- Fixed a bug where creating an `HDUList` object containing multiple PRIMARY HDUs caused an infinite recursion when validating the object prior to writing to a file. (#145)
- Fixed a rare but serious case where saving an update to a file that previously had a CHECKSUM and/or DATASUM keyword, but removed the checksum in saving, could cause the file to be slightly corrupted and unreadable. (#147)
- Fixed problems with reading "non-standard" FITS files with primary headers containing SIMPLE = F. PyFITS has never made many guarantees as to how such files are handled. But it should at least be possible to read their

headers, and the data if possible. Saving changes to such a file should not try to prepend an unwanted valid PRIMARY HDU. (#157)

- Fixed a bug where opening an image with `disable_image_compression = True` caused compression to be disabled for all subsequent `pyfits.open()` calls. (r1651)

3.0.7 (2012-04-10)

Changes in Behavior

- Slices of GroupData objects now return new GroupData objects instead of extended multi-row _Group objects. This is analogous to how PyFITS 3.0 fixed FITS_rec slicing, and should have been fixed for GroupData at the same time. The old behavior caused bugs where functions internal to Numpy expected that slicing an ndarray would return a new ndarray. As this is a rare use case with a rare feature most users are unlikely to be affected by this change.
- The previously internal _Group object for representing individual group records in a GroupData object are renamed Group and are now a public interface. However, there's almost no good reason to create Group objects directly, so it shouldn't be considered a "new feature".
- An annoyance from PyFITS 3.0.6 was fixed, where the value of the EXTEND keyword was always being set to F if there are not actually any extension HDUs. It was unnecessary to modify this value.

Bug Fixes

- Fixed GroupData objects to return new GroupData objects when sliced instead of _Group record objects. See "Changes in behavior" above for more details.
- Fixed slicing of Group objects–previously it was not possible to slice slice them at all.
- Made it possible to assign `np.bool_` objects as header values. (#123)
- Fixed overly strict handling of the EXTEND keyword; see "Changes in behavior" above. (#124)
- Fixed many cases where an HDU's header would be marked as "modified" by PyFITS and rewritten, even when no changes to the header are necessary. (#125)
- Fixed a bug where the values of the PTYPEn keywords in a random groups HDU were forced to be all lower-case when saving the file. (#130)
- Removed an unnecessary inline import in `ExtensionHDU.__setattr__` that was causing some slowdown when opening files containing a large

number of extensions, plus a few other small (but not insignificant) performance improvements thanks to Julian Taylor. (#133)
- Fixed a regression where header blocks containing invalid end-of-header padding (i.e. null bytes instead of spaces) couldn't be parsed by PyFITS. Such headers can be parsed again, but a warning is raised, as such headers are not valid FITS. (#136)
- Fixed a memory leak where table data in random groups HDUs weren't being garbage collected. (#138)

3.0.6 (2012-02-29)

Highlights

The main reason for this release is to fix an issue that was introduced in PyFITS 3.0.5 where merely opening a file containing scaled data (that is, with non-trivial BSCALE and BZERO keywords) in 'update' mode would cause the data to be automatically rescaled–possibly converting the data from ints to floats–as soon as the file is closed, even if the application did not touch the data. Now PyFITS will only rescale the data in an extension when the data is actually accessed by the application. So opening a file in 'update' mode in order to modify the header or append new extensions will not cause any change to the data in existing extensions.

This release also fixes a few Windows-specific bugs found through more extensive Windows testing, and other miscellaneous bugs.

Bug Fixes

- More accurate error messages when opening files containing invalid header cards. (#109)
- Fixed a possible reference cycle/memory leak that was caught through more extensive testing on Windows. (#112)
- Fixed 'ostream' mode to open the underlying file in 'wb' mode instead of 'w' mode. (#112)
- Fixed a Windows-only issue where trying to save updates to a resized FITS file could result in a crash due to there being open mmaps on that file. (#112)
- Fixed a crash when trying to create a FITS table (i.e. with new_table()) from a Numpy array containing bool fields. (#113)
- Fixed a bug where manually initializing an `HDUList` with a list of of HDUs wouldn't set the correct EXTEND keyword value on the primary HDU. (#114)
- Fixed a crash that could occur when trying to deepcopy a Header in Python < 2.7. (#115)
- Fixed an issue where merely opening a scaled image in 'update' mode would cause the data to be converted to floats when the file is closed. (#119)

3.0.5 (2012-01-30)

- Fixed a crash that could occur when accessing image sections of files opened with memmap=True. (r1211)
- Fixed the inconsistency in the behavior of files opened in 'readonly' mode when memmap=True vs. when memmap=False. In the latter case, although changes to array data were not saved to disk, it was possible to update the array data in memory. On the other hand with memmap=True, 'readonly' mode prevented even in-memory modification to the data. This is what 'copyonwrite' mode was for, but difference in behavior was confusing. Now 'readonly' is equivalent to 'copyonwrite' when using memmap. If the old behavior of denying changes to the array data is necessary, a new 'denywrite' mode may be used, though it is only applicable to files opened with memmap. (r1275)
- Fixed an issue where files opened with memmap=True would return image data as a raw numpy.memmap object, which can cause some unexpected behaviors–instead memmap object is viewed as a numpy.ndarray. (r1285)
- Fixed an issue in Python 3 where a workaround for a bug in Numpy on Python 3 interacted badly with some other software, namely to vo.table package (and possibly others). (r1320, r1337, and #110)
- Fixed buggy behavior in the handling of SIGINTs (i.e. Ctrl-C keyboard interrupts) while flushing changes to a FITS file. PyFITS already prevented SIGINTs from causing an incomplete flush, but did not clean up the signal handlers properly afterwards, or reraise the keyboard interrupt once the flush was complete. (r1321)
- Fixed a crash that could occur in Python 3 when opening files with checksum checking enabled. (r1336)
- Fixed a small bug that could cause a crash in the `StreamingHDU` interface when using Numpy below version 1.5.
- Fixed a crash that could occur when creating a new `CompImageHDU` from an array of big-endian data. (#104)
- Fixed a crash when opening a file with extra zero padding at the end. Though FITS files should not have such padding, it's not explicitly forbidden by the format either, and PyFITS shouldn't stumble over it. (#106)
- Fixed a major slowdown in opening tables containing large columns of string values. (#111)

3.0.4 (2011-11-22)

- Fixed a crash when writing HCOMPRESS compressed images that could happen on Python 2.5 and 2.6. (r1217)

- Fixed a crash when slicing an table in a file opened in 'readonly' mode with memmap=True. (r1230)
- Writing changes to a file or writing to a new file verifies the output in 'fix' mode by default instead of 'exception'–that is, PyFITS will automatically fix common FITS format errors rather than raising an exception. (r1243)
- Fixed a bug where convenience functions such as getval() and getheader() crashed when specifying just 'PRIMARY' as the extension to use (r1263).
- Fixed a bug that prevented passing keyword arguments (beyond the standard data and header arguments) as positional arguments to the constructors of extension HDU classes.
- Fixed some tests that were failing on Windows–in this case the tests themselves failed to close some temp files and Windows refused to delete them while there were still open handles on them. (r1295)
- Fixed an issue with floating point formatting in header values on Python 2.5 for Windows (and possibly other platforms). The exponent was zero-padded to 3 digits; although the FITS standard makes no specification on this, the formatting is now normalized to always pad the exponent to two digits. (r1295)
- Fixed a bug where long commentary cards (such as HISTORY and COMMENT) were broken into multiple CONTINUE cards. However, commentary cards are not expected to be found in CONTINUE cards. Instead these long cards are broken into multiple commentary cards. (#97)
- GZIP/ZIP-compressed FITS files can be detected and opened regardless of their filename extension. (#99)
- Fixed a serious bug where opening scaled images in 'update' mode and then closing the file without touching the data would cause the file to be corrupted. (#101)

3.0.3 (2011-10-05)

- Fixed several small bugs involving corner cases in record-valued keyword cards (#70)
- In some cases HDU creation failed if the first keyword value in the header was not a string value (#89)
- Fixed a crash when trying to compute the HDU checksum when the data array contains an odd number of bytes (#91)
- Disabled an unnecessary warning that was displayed on opening compressed HDUs with disable_image_compression = True (#92)
- Fixed a typo in code for handling HCOMPRESS compressed images.

3.0.2 (2011-09-23)

- The `BinTableHDU.tcreate` method and by extension the `pyfits.tcreate` function don't get tripped up by blank lines anymore (#14)
- The presence, value, and position of the EXTEND keyword in Primary HDUs is verified when reading/writing a FITS file (#32)
- Improved documentation (in warning messages as well as in the handbook) that PyFITS uses zero-based indexing (as one would expect for C/Python code, but contrary to the PyFITS standard which was written with FORTRAN in mind) (#68)
- Fixed a bug where updating a header card comment could cause the value to be lost if it had not already been read from the card image string.
- Fixed a related bug where changes made directly to Card object in a header (i.e. assigning directly to card.value or card.comment) would not propagate when flushing changes to the file (#69) [Note: This and the bug above it were originally reported as being fixed in version 3.0.1, but the fix was never included in the release.]
- Improved file handling, particularly in Python 3 which had a few small file I/O-related bugs (#76)
- Fixed a bug where updating a FITS file would sometimes cause it to lose its original file permissions (#79)
- Fixed the handling of TDIMn keywords; 3.0 added support for them, but got the axis order backwards (they were treated as though they were row-major) (#82)
- Fixed a crash when a FITS file containing scaled data is opened and immediately written to a new file without explicitly viewing the data first (#84)
- Fixed a bug where creating a table with columns named either 'names' or 'formats' resulted in an infinite recursion (#86)

3.0.1 (2011-09-12)

- Fixed a bug where updating a header card comment could cause the value to be lost if it had not already been read from the card image string.
- Changed `_TableBaseHDU.data` so that if the data contain an empty table a `FITS_rec` object with zero rows is returned rather than `None`.
- The `.key` attribute of `RecordValuedKeywordCards` now returns the full keyword+field-specifier value, instead of just the plain keyword (#46)
- Fixed a related bug where changes made directly to Card object in a header (i.e. assigning directly to card.value or card.comment) would not propagate when flushing changes to the file (#69)

- Fixed a bug where writing a table with zero rows could fail in some cases (#72)
- Miscellaneous small bug fixes that were causing some tests to fail, particularly on Python 3 (#74, #75)
- Fixed a bug where creating a table column from an array in non-native byte order would not preserve the byte order, thus interpreting the column array using the wrong byte order (#77)

3.0.0 (2011-08-23)

- Contains major changes, bumping the version to 3.0
- Large amounts of refactoring and reorganization of the code; tried to preserve public API backwards-compatibility with older versions (private API has many changes and is not guaranteed to be backwards-compatible). There are a few small public API changes to be aware of:
    - The pyfits.rec module has been removed completely. If your version of numpy does not have the numpy.core.records module it is too old to be used with PyFITS.
    - The `Header.ascardlist()` method is deprecated–use the `.ascard` attribute instead.
    - `Card` instances have a new `.cardimage` attribute that should be used rather than `.ascardimage()`, which may become deprecated.
    - The `Card.fromstring()` method is now a classmethod. It returns a new `Card` instance rather than modifying an existing instance.
    - The `req_cards()` method on HDU instances has changed: The `pos` argument is not longer a string. It is either an integer value (meaning the card's position must match that value) or it can be a function that takes the card's position as it's argument, and returns True if the position is valid. Likewise, the `test` argument no longer takes a string, but instead a function that validates the card's value and returns True or False.
    - The `get_coldefs()` method of table HDUs is deprecated. Use the `.columns` attribute instead.
    - The `ColDefs.data` attribute is deprecated–use `ColDefs.columns` instead (though in general you shouldn't mess with it directly–it might become internal at some point).
    - `FITS_record` objects take `start` and `end` as arguments instead of `startColumn` and `endColumn` (these are rarely created manually, so it's unlikely that this change will affect anyone).
    - `BinTableHDU.tcreate()` is now a classmethod, and returns a new `BinTableHDU` instance.

  - Use `ExtensionHDU` and `NonstandardExtHDU` for making new extension HDU classes. They are now public interfaces, wheres previously they were private and prefixed with underscores.
  - Possibly others–please report if you find any changes that cause difficulties.
- Calls to deprecated functions will display a Deprecation warning. However, in Python 2.7 and up Deprecation warnings are ignored by default, so run Python with the `-Wd` option to see if you're using any deprecated functions. If we get close to actually removing any functions, we might make the Deprecation warnings display by default.
- Added basic Python 3 support
- Added support for multi-dimensional columns in tables as specified by the TDIMn keywords (#47)
- Fixed a major memory leak that occurred when creating new tables with the `new_table()` function (#49) be padded with zero-bytes) vs ASCII tables (where strings are padded with spaces) (#15)
- Fixed a bug in which the case of Random Access Group parameters names was not preserved when writing (#41)
- Added support for binary table fields with zero width (#42)
- Added support for wider integer types in ASCII tables; although this is non-standard, some GEIS images require it (#45)
- Fixed a bug that caused the index_of() method of HDULists to crash when the HDUList object is created from scratch (#48)
- Fixed the behavior of string padding in binary tables (where strings should be padded with nulls instead of spaces)
- Fixed a rare issue that caused excessive memory usage when computing checksums using a non-standard block size (see r818)
- Add support for forced uint data in image sections (#53)
- Fixed an issue where variable-length array columns were not extended when creating a new table with more rows than the original (#54)
- Fixed tuple and list-based indexing of FITS_rec objects (#55)
- Fixed an issue where BZERO and BSCALE keywords were appended to headers in the wrong location (#56)
- `FITS_record` objects (table rows) have full slicing support, including stepping, etc. (#59)
- Fixed a bug where updating multiple files simultaneously (such as when running parallel processes) could lead to a race condition with mktemp() (#61)
- Fixed a bug where compressed image headers were not in the order expected by the funpack utility (#62)

2.4.0 (2011-01-10)

The following enhancements were added:

- Checksum support now correctly conforms to the FITS standard. pyfits supports reading and writing both the old checksums and new standard-compliant checksums. The `fitscheck` command-line utility is provided to verify and update checksums.
- Added a new optional keyword argument `do_not_scale_image_data` to the `pyfits.open` convenience function. When this argument is provided as True, and an ImageHDU is read that contains scaled data, the data is not automatically scaled when it is read. This option may be used when opening a fits file for update, when you only want to update some header data. Without the use of this argument, if the header updates required the size of the fits file to change, then when writing the updated information, the data would be read, scaled, and written back out in its scaled format (usually with a different data type) instead of in its non-scaled format.
- Added a new optional keyword argument `disable_image_compression` to the `pyfits.open` function. When `True`, any compressed image HDU's will be read in like they are binary table HDU's.
- Added a `verify` keyword argument to the `pyfits.append` function. When `False`, `append` will assume the existing FITS file is already valid and simply append new content to the end of the file, resulting in a large speed up appending to large files.
- Added HDU methods `update_ext_name` and `update_ext_version` for updating the name and version of an HDU.
- Added HDU method `filebytes` to calculate the number of bytes that will be written to the file associated with the HDU.
- Enhanced the section class to allow reading non-contiguous image data. Previously, the section class could only be used to read contiguous data. (CNSHD781626)
- Added method `HDUList.fileinfo()` that returns a dictionary with information about the location of header and data in the file associated with the HDU.

The following bugs were fixed:

- Reading in some malformed FITS headers would cause a `NameError` exception, rather than information about the cause of the error.
- pyfits can now handle non-compliant `CONTINUE` cards produced by Java FITS.
- `BinTable` columns with `TSCALn` are now byte-swapped correctly.
- Ensure that floating-point card values are no longer than 20 characters.
- Updated `flush` so that when the data has changed in an HDU for a file

opened in update mode, the header will be updated to match the changed data before writing out the HDU.
- Allow `HIERARCH` cards to contain a keyword and value whose total character length is 69 characters. Previous length was limited at 68 characters.
- Calls to `FITS_rec['columnName']` now return an `ndarray`. exactly the same as a call to `FITS_rec.field('columnName')` or `FITS_rec.columnName`. Previously, `FITS_rec['columnName']` returned a much less useful `fits_record` object. (CNSHD789053)
- Corrected the `append` convenience function to eliminate the reading of the HDU data from the file that is being appended to. (CNSHD794738)
- Eliminated common symbols between the pyfitsComp module and the cfitsio and zlib libraries. These can cause problems on systems that use both PyFITS and cfitsio or zlib. (CNSHD795046)

2.3.1 (2010-06-03)

The following bugs were fixed:

- Replaced code in the Compressed Image HDU extension which was covered under a GNU General Public License with code that is covered under a BSD License. This change allows the distribution of pyfits under a BSD License.

2.3 (2010-05-11)

The following enhancements were made:

- Completely eliminate support for numarray.
- Rework pyfits documentation to use Sphinx.
- Support python 2.6 and future division.
- Added a new method to get the file name associated with an HDUList object. The method HDUList.filename() returns the name of an associated file. It returns None if no file is associated with the HDUList.
- Support the python 2.5 ‘with’ statement when opening fits files. (CNSHD766308) It is now possible to use the following construct:

```
>>> from __future__ import with_statement import pyfits
>>> with pyfits.open("input.fits") as hdul:
```

```
...     #process hdul
>>>
```

- Extended the support for reading unsigned integer 16 values from an ImageHDU to include unsigned integer 32 and unsigned integer 64 values. ImageHDU data is considered to be unsigned integer 16 when the data type is signed integer 16 and BZERO is equal to 2**15 (32784) and BSCALE is equal to 1. ImageHDU data is considered to be unsigned integer 32 when the data type is signed integer 32 and BZERO is equal to 2**31 and BSCALE is equal to 1. ImageHDU data is considered to be unsigned integer 64 when the data type is signed integer 64 and BZERO is equal to 2**63 and BSCALE is equal to 1. An optional keyword argument (uint) was added to the open convenience function for this purpose. Supplying a value of True for this argument will cause data of any of these types to be read in and scaled into the appropriate unsigned integer array (uint16, uint32, or uint64) instead of into the normal float 32 or float 64 array. If an HDU associated with a file that was opened with the 'int' option and containing unsigned integer 16, 32, or 64 data is written to a file, the data will be reverse scaled into a signed integer 16, 32, or 64 array and written out to the file along with the appropriate BSCALE/BZERO header cards. Note that for backward compatibility, the 'uint16' keyword argument will still be accepted in the open function when handling unsigned integer 16 conversion.
- Provided the capability to access the data for a column of a fits table by indexing the table using the column name. This is consistent with Record Arrays in numpy (array with fields). (CNSHD763378) The following example will illustrate this:

```
>>> import pyfits
>>> hdul = pyfits.open('input.fits')
>>> table = hdul[1].data
>>> table.names
['c1','c2','c3','c4']
>>> print table.field('c2') # this is the data for column 2
['abc' 'xy']
>>> print table['c2'] # this is also the data for column 2
array(['abc', 'xy '], dtype='|S3')
>>> print table[1] # this is the data for row 1
(2, 'xy', 6.6999997138977054, True)
```

- Provided capabilities to create a BinaryTableHDU directly from a numpy Record Array (array with fields). The new capabilities include table creation, writing a numpy Record Array directly to a fits file using the pyfits.writeto and pyfits.append convenience functions. Reading the data for a BinaryTableHDU from a fits file directly into a numpy Record Array using the pyfits.getdata convenience function. (CNSHD749034) Thanks to Erin Sheldon at Brookhaven National Laboratory for help with this.

The following should illustrate these new capabilities:

```
>>> import pyfits
>>> import numpy
>>> t=numpy.zeros(5,dtype=[('x','f4'),('y','2i4')]) \
... # Create a numpy Record Array with fields
>>> hdu = pyfits.BinTableHDU(t) \
... # Create a Binary Table HDU directly from the Record Array
>>> print hdu.data
[(0.0, array([0, 0], dtype=int32))
 (0.0, array([0, 0], dtype=int32))
 (0.0, array([0, 0], dtype=int32))
 (0.0, array([0, 0], dtype=int32))
 (0.0, array([0, 0], dtype=int32))]
>>> hdu.writeto('test1.fits',clobber=True) \
... # Write the HDU to a file
>>> pyfits.info('test1.fits')
Filename: test1.fits
No.    Name         Type      Cards   Dimensions   Format
0    PRIMARY     PrimaryHDU       4   ()           uint8
1                BinTableHDU     12   5R x 2C      [E, 2J]
>>> pyfits.writeto('test.fits', t, clobber=True) \
... # Write the Record Array directly to a file
>>> pyfits.append('test.fits', t) \
... # Append another Record Array to the file
>>> pyfits.info('test.fits')
Filename: test.fits
No.    Name         Type      Cards   Dimensions   Format
0    PRIMARY     PrimaryHDU       4   ()           uint8
1                BinTableHDU     12   5R x 2C      [E, 2J]
2                BinTableHDU     12   5R x 2C      [E, 2J]
>>> d=pyfits.getdata('test.fits',ext=1) \
... # Get the first extension from the file as a FITS_rec
>>> print type(d)
<class 'pyfits.core.FITS_rec'>
>>> print d
[(0.0, array([0, 0], dtype=int32))
 (0.0, array([0, 0], dtype=int32))
 (0.0, array([0, 0], dtype=int32))
 (0.0, array([0, 0], dtype=int32))
 (0.0, array([0, 0], dtype=int32))]
>>> d=pyfits.getdata('test.fits',ext=1,view=numpy.ndarray) \
... # Get the first extension from the file as a numpy Record
      Array
>>> print type(d)
<type 'numpy.ndarray'>
```

```
>>> print d
[(0.0, [0, 0]) (0.0, [0, 0]) (0.0, [0, 0]) (0.0, [0, 0])
 (0.0, [0, 0])]
>>> print d.dtype
[('x', '>f4'), ('y', '>i4', 2)]
>>> d=pyfits.getdata('test.fits',ext=1,upper=True,
...                  view=pyfits.FITS_rec) \
... # Force the Record Array field names to be in upper case
      regardless of how they are stored in the file
>>> print d.dtype
[('X', '>f4'), ('Y', '>i4', 2)]
```

- Provided support for writing fits data to file-like objects that do not support the random access methods seek() and tell(). Most pyfits functions or methods will treat these file-like objects as an empty file that cannot be read, only written. It is also expected that the file-like object is in a writable condition (ie. opened) when passed into a pyfits function or method. The following methods and functions will allow writing to a non-random access file-like object: HDUList.writeto(), HDUList.flush(), pyfits.writeto(), and pyfits.append(). The pyfits.open() convenience function may be used to create an HDUList object that is associated with the provided file-like object. (CNSHD770036)

  An illustration of the new capabilities follows. In this example fits data is written to standard output which is associated with a file opened in write-only mode:

```
>>> import pyfits
>>> import numpy as np
>>> import sys
>>>
>>> hdu = pyfits.PrimaryHDU(np.arange(100,dtype=np.int32))
>>> hdul = pyfits.HDUList()
>>> hdul.append(hdu)
>>> tmpfile = open('tmpfile.py','w')
>>> sys.stdout = tmpfile
>>> hdul.writeto(sys.stdout, clobber=True)
>>> sys.stdout = sys.__stdout__
>>> tmpfile.close()
>>> pyfits.info('tmpfile.py')
Filename: tmpfile.py
No.    Name         Type      Cards   Dimensions   Format
0    PRIMARY     PrimaryHDU       5   (100,)       int32
>>>
```

- Provided support for slicing a FITS_record object. The FITS_record object represents the data from a row of a table. Pyfits now supports the slice syntax to retrieve values from the row. The following illustrates this new syntax:

```
>>> hdul = pyfits.open('table.fits')
>>> row = hdul[1].data[0]
>>> row
('clear', 'nicmos', 1, 30, 'clear', 'idno= 100')
>>> a, b, c, d, e = row[0:5]
>>> a
'clear'
>>> b
'nicmos'
>>> c
1
>>> d
30
>>> e
'clear'
>>>
```

- Allow the assignment of a row value for a pyfits table using a tuple or a list as input. The following example illustrates this new feature:

```
>>> c1=pyfits.Column(name='target',format='10A')
>>> c2=pyfits.Column(name='counts',format='J',unit='DN')
>>> c3=pyfits.Column(name='notes',format='A10')
>>> c4=pyfits.Column(name='spectrum',format='5E')
>>> c5=pyfits.Column(name='flag',format='L')
>>> coldefs=pyfits.ColDefs([c1,c2,c3,c4,c5])
>>>
>>> tbhdu=pyfits.new_table(coldefs, nrows = 5)
>>>
>>> # Assigning data to a table's row using a tuple
>>> tbhdu.data[2] = ('NGC1',312,'A Note',
... num.array([1.1,2.2,3.3,4.4,5.5],dtype=num.float32),
... True)
>>>
>>> # Assigning data to a tables row using a list
>>> tbhdu.data[3] = ['JIM1','33','A Note',
... num.array([1.,2.,3.,4.,5.],dtype=num.float32),True]
```

- Allow the creation of a Variable Length Format (P format) column from a list of data. The following example illustrates this new feature:

```
>>> a = [num.array([7.2e-20,7.3e-20]),num.array([0.0]),
... num.array([0.0])]
>>> acol = pyfits.Column(name='testa',format='PD()',array=a)
>>> acol.array
_VLF([[  7.20000000e-20   7.30000000e-20], [ 0.], [ 0.]],
dtype=object)
>>>
```

- Allow the assignment of multiple rows in a table using the slice syntax. The following example illustrates this new feature:

```
>>> counts = num.array([312,334,308,317])
>>> names = num.array(['NGC1','NGC2','NGC3','NCG4'])
>>> c1=pyfits.Column(name='target',format='10A',array=names)
>>> c2=pyfits.Column(name='counts',format='J',unit='DN',
... array=counts)
>>> c3=pyfits.Column(name='notes',format='A10')
>>> c4=pyfits.Column(name='spectrum',format='5E')
>>> c5=pyfits.Column(name='flag',format='L',array=[1,0,1,1])
>>> coldefs=pyfits.ColDefs([c1,c2,c3,c4,c5])
>>>
>>> tbhdu1=pyfits.new_table(coldefs)
>>>
>>> counts = num.array([112,134,108,117])
>>> names = num.array(['NGC5','NGC6','NGC7','NCG8'])
>>> c1=pyfits.Column(name='target',format='10A',array=names)
>>> c2=pyfits.Column(name='counts',format='J',unit='DN',
... array=counts)
>>> c3=pyfits.Column(name='notes',format='A10')
>>> c4=pyfits.Column(name='spectrum',format='5E')
>>> c5=pyfits.Column(name='flag',format='L',array=[0,1,0,0])
>>> coldefs=pyfits.ColDefs([c1,c2,c3,c4,c5])
>>>
>>> tbhdu=pyfits.new_table(coldefs)
>>> tbhdu.data[0][3] = num.array([1.,2.,3.,4.,5.],
... dtype=num.float32)
>>>
>>> tbhdu2=pyfits.new_table(tbhdu1.data, nrows=9)
>>>
>>> # Assign the 4 rows from the second table to rows 5 thru
...   8 of the new table.  Note that the last row of the new
...   table will still be initialized to the default values.
>>> tbhdu2.data[4:] = tbhdu.data
>>>
>>> print tbhdu2.data
[ ('NGC1', 312, '0.0', array([ 0.,  0.,  0.,  0.,  0.],
dtype=float32), True)
  ('NGC2', 334, '0.0', array([ 0.,  0.,  0.,  0.,  0.],
dtype=float32), False)
  ('NGC3', 308, '0.0', array([ 0.,  0.,  0.,  0.,  0.],
dtype=float32), True)
```

```
  ('NCG4', 317, '0.0', array([ 0.,  0.,  0.,  0.,  0.],
dtype=float32), True)
  ('NGC5', 112, '0.0', array([ 1.,  2.,  3.,  4.,  5.],
dtype=float32), False)
  ('NGC6', 134, '0.0', array([ 0.,  0.,  0.,  0.,  0.],
dtype=float32), True)
  ('NGC7', 108, '0.0', array([ 0.,  0.,  0.,  0.,  0.],
dtype=float32), False)
  ('NCG8', 117, '0.0', array([ 0.,  0.,  0.,  0.,  0.],
dtype=float32), False)
  ('0.0', 0, '0.0', array([ 0.,  0.,  0.,  0.,  0.],
dtype=float32), False)]
>>>
```

The following bugs were fixed:

- Corrected bugs in HDUList.append and HDUList.insert to correctly handle the situation where you want to insert or append a Primary HDU as something other than the first HDU in an HDUList and the situation where you want to insert or append an Extension HDU as the first HDU in an HDUList.
- Corrected a bug involving scaled images (both compressed and not compressed) that include a BLANK, or ZBLANK card in the header. When the image values match the BLANK or ZBLANK value, the value should be replaced with NaN after scaling. Instead, pyfits was scaling the BLANK or ZBLANK value and returning it. (CNSHD766129)
- Corrected a byteswapping bug that occurs when writing certain column data. (CNSHD763307)
- Corrected a bug that occurs when creating a column from a chararray when one or more elements are shorter than the specified format length. The bug wrote nulls instead of spaces to the file. (CNSHD695419)
- Corrected a bug in the HDU verification software to ensure that the header contains no NAXISn cards where n > NAXIS.
- Corrected a bug involving reading and writing compressed image data. When written, the header keyword card ZTENSION will always have the value 'IMAGE' and when read, if the ZTENSION value is not 'IMAGE' the user will receive a warning, but the data will still be treated as image data.
- Corrected a bug that restricted the ability to create a custom HDU class and use it with pyfits. The bug fix will allow something like this:

```
>>> import pyfits
>>> class MyPrimaryHDU(pyfits.PrimaryHDU):
...     def __init__(self, data=None, header=None):
...         pyfits.PrimaryHDU.__init__(self, data, header)
...     def _summary(self):
...         """
...         Reimplement a method of the class.
...         """
...         s = pyfits.PrimaryHDU._summary(self)
...         # change the behavior to suit me.
...         s1 = 'MyPRIMARY ' + s[11:]
...         return s1
...
>>> hdul=pyfits.open("pix.fits",
... classExtensions={pyfits.PrimaryHDU: MyPrimaryHDU})
>>> hdul.info()
Filename: pix.fits
No.    Name         Type      Cards   Dimensions   Format
0    MyPRIMARY  MyPrimaryHDU     59  (512, 512)    int16
>>>
```

- Modified ColDefs.add_col so that instead of returning a new ColDefs object with the column added to the end, it simply appends the new column to the current ColDefs object in place. (CNSHD768778)
- Corrected a bug in ColDefs.del_col which raised a KeyError exception when deleting a column from a ColDefs object.
- Modified the open convenience function so that when a file is opened in readonly mode and the file contains no HDU's an IOError is raised.
- Modified _TableBaseHDU to ensure that all locations where data is referenced in the object actually reference the same ndarray, instead of copies of the array.
- Corrected a bug in the Column class that failed to initialize data when the data is a boolean array. (CNSHD779136)
- Corrected a bug that caused an exception to be raised when creating a variable length format column from character data (PA format).
- Modified installation code so that when installing on Windows, when a C++ compiler compatible with the Python binary is not found, the installation completes with a warning that all optional extension modules failed to build. Previously, an Error was issued and the installation stopped.

2.2.2 (2009-10-12)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following bugs were fixed:

- Corrected a bug that caused an exception to be raised when creating a CompImageHDU using an initial header that does not match the image data in terms of the number of axis.

2.2.1 (2009-10-06)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following bugs were fixed:

- Corrected a bug that prevented the opening of a fits file where a header contained a CHECKSUM card but no DATASUM card.
- Corrected a bug that caused NULLs to be written instead of blanks when an ASCII table was created using a numpy chararray in which the original data contained trailing blanks. (CNSHD695419)

2.2 (2009-09-23)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following enhancements were made:

- Provide support for the FITS Checksum Keyword Convention. (CNSHD754301)
- Adding the checksum=True keyword argument to the open convenience function will cause checksums to be verified on file open:

```
>>> hdul=pyfits.open('in.fits', checksum=True)
```

- On output, CHECKSUM and DATASUM cards may be output to all HDU's in a fits file by using the keyword argument checksum=True in calls to the writeto convenience function, the HDUList.writeto method, the writeto methods of all of the HDU classes, and the append convenience function:

```
>>> hdul.writeto('out.fits', checksum=True)
```

- Implemented a new insert method to the HDUList class that allows for the insertion of a HDU into a HDUList at a given index:

```
>>> hdul.insert(2,hdu)
```

- Provided the capability to handle Unicode input for file names.
- Provided support for integer division required by Python 3.0.

The following bugs were fixed:

- Corrected a bug that caused an index out of bounds exception to be raised when iterating over the rows of a binary table HDU using the syntax “for row in tbhdu.data: “. (CNSHD748609)
- Corrected a bug that prevented the use of the writeto convenience function for writing table data to a file. (CNSHD749024)
- Modified the code to raise an IOError exception with the comment “Header missing END card.” when pyfits can’t find a valid END card for a header when opening a file.
  - This change addressed a problem with a non-standard fits file that contained several new-line characters at the end of each header and at the end of the file. However, since some people want to be able to open these non-standard files anyway, an option was added to the open convenience function to allow these files to be opened without exception:

    ```
    >>> pyfits.open('infile.fits',ignore_missing_end=True)
    ```

- Corrected a bug that prevented the use of StringIO objects as fits files when reading and writing table data. Previously, only image data was supported. (CNSHD753698)
- Corrected a bug that caused a bus error to be generated when compressing image data using GZIP_1 under the Solaris operating system.
- Corrected bugs that prevented pyfits from properly reading Random Groups HDU’s using numpy. (CNSHD756570)
- Corrected a bug that can occur when writing a fits file. (CNSHD757508)
  - If no default SIGINT signal handler has not been assigned, before the write, a TypeError exception is raised in the _File.flush() method when attempting to return the signal handler to its previous state. Notably this occurred when using mod_python. The code was changed to use SIG_DFL when no old handler was defined.
- Corrected a bug in CompImageHDU that prevented rescaling the image data using hdu.scale(option=’old’).

2.1.1 (2009-04-22)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following bugs were fixed:

- Corrected a bug that caused an exception to be raised when closing a file opened for append, where an HDU was appended to the file, after data was accessed from the file. This exception was only raised when running on a Windows platform.
- Updated the installation scripts, compression source code, and benchmark test scripts to properly install, build, and execute on a Windows platform.

2.1 (2009-04-14)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following enhancements were made:

- Added new tdump and tcreate capabilities to pyfits.
  - The new tdump convenience function allows the contents of a binary table HDU to be dumped to a set of three files in ASCII format. One file will contain column definitions, the second will contain header parameters, and the third will contain header data.
  - The new tcreate convenience function allows the creation of a binary table HDU from the three files dumped by the tdump convenience function.
  - The primary use for the tdump/tcreate methods are to allow editing in a standard text editor of the binary table data and parameters.
- Added support for case sensitive values of the EXTNAME card in an extension header. (CNSHD745784)
  - By default, pyfits converts the value of EXTNAME cards to upper case when reading from a file. A new convenience function (setExtensionNameCaseSensitive) was implemented to allow a user to circumvent this behavior so that the EXTNAME value remains in the same case as it is in the file.

  - With the following function call, pyfits will maintain the case of all characters in the EXTNAME card values of all extension HDU’s during the entire python session, or until another call to the function is made:

```
>>> import pyfits
>>> pyfits.setExtensionNameCaseSensitive()
```

- The following function call will return pyfits to its default (all upper case) behavior:

  ```
  >>> pyfits.setExtensionNameCaseSensitive(False)
  ```

- Added support for reading and writing FITS files in which the value of the first card in the header is ‘SIMPLE=F’. In this case, the pyfits open function returns an HDUList object that contains a single HDU of the new type _NonstandardHDU. The header for this HDU is like a normal header (with the exception that the first card contains SIMPLE=F instead of SIMPLE=T). Like normal HDU’s the reading of the data is delayed until actually requested. The data is read from the file into a string starting from the first byte after the header END card and continuing till the end of the file. When written, the header is written, followed by the data string. No attempt is made to pad the data string so that it fills into a standard 2880 byte FITS block. (CNSHD744730)
- Added support for FITS files containing extensions with unknown XTENSION card values. (CNSHD744730) Standard FITS files support extension HDU’s of types TABLE, IMAGE, BINTABLE, and A3DTABLE. Accessing a nonstandard extension from a FITS file will now create a _NonstandardExtHDU object. Accessing the data of this object will cause the data to be read from the file into a string. If the HDU is written back to a file the string data is written after the Header and padded to fill a standard 2880 byte FITS block.

The following bugs were fixed:

- Extensive changes were made to the tiled image compression code to support the latest enhancements made in CFITSIO version 3.13 to support this convention.
- Eliminated a memory leak in the tiled image compression code.
- Corrected a bug in the FITS_record.__setitem__ method which raised a NameError exception when attempting to set a value in a FITS_record object. (CNSHD745844)
- Corrected a bug that caused a TypeError exception to be raised when reading fits files containing large table HDU’s (>2Gig). (CNSHD745522)
- Corrected a bug that caused a TypeError exception to be raised for all calls to the warnings module when running under Python 2.6. The formatwarning method in the warnings module was changed in Python 2.6 to include a new argument. (CNSHD746592)
- Corrected the behavior of the membership (in) operator in the Header class to check against header card keywords instead of card values. (CNSHD744730)
- Corrected the behavior of iteration on a Header object. The new behavior

iterates over the unique card keywords instead of the card values.

2.0.1 (2009-02-03)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following bugs were fixed:

- Eliminated a memory leak when reading Table HDU's from a fits file. (CNSHD741877)

2.0 (2009-01-30)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following enhancements were made:

- Provide initial support for an image compression convention known as the "Tiled Image Compression Convention" [1].
  - The principle used in this convention is to first divide the n-dimensional image into a rectangular grid of subimages or "tiles". Each tile is then compressed as a continuous block of data, and the resulting compressed byte stream is stored in a row of a variable length column in a FITS binary table. Several commonly used algorithms for compressing image tiles are supported. These include, GZIP, RICE, H-Compress and IRAF pixel list (PLIO).
  - Support for compressed image data is provided using the optional "pyfitsComp" module contained in a C shared library (pyfitsCompmodule.so).
  - The header of a compressed image HDU appears to the user like any image header. The actual header stored in the FITS file is that of a binary table HDU with a set of special keywords, defined by the convention, to describe the structure of the compressed image. The conversion between binary table HDU header and image HDU header is all performed behind the scenes. Since the HDU is actually a binary table, it may not appear as a primary HDU in a FITS file.
  - The data of a compressed image HDU appears to the user as standard uncompressed image data. The actual data is stored in the fits file as Binary Table data containing at least one column (COMPRESSED_DATA).

Each row of this variable-length column contains the byte stream that was generated as a result of compressing the corresponding image tile. Several optional columns may also appear. These include, UNCOMPRESSED_DATA to hold the uncompressed pixel values for tiles that cannot be compressed, ZSCALE and ZZERO to hold the linear scale factor and zero point offset which may be needed to transform the raw uncompressed values back to the original image pixel values, and ZBLANK to hold the integer value used to represent undefined pixels (if any) in the image.

- To create a compressed image HDU from scratch, simply construct a CompImageHDU object from an uncompressed image data array and its associated image header. From there, the HDU can be treated just like any image HDU:

```
>>> hdu=pyfits.CompImageHDU(imageData,imageHeader)
>>> hdu.writeto('compressed_image.fits')
```

- The signature for the CompImageHDU initializer method describes the possible options for constructing a CompImageHDU object:

```
def __init__(self, data=None, header=None, name=None,
             compressionType='RICE_1',
             tileSize=None,
             hcompScale=0.,
             hcompSmooth=0,
             quantizeLevel=16.):
    """
        data:            data of the image
        header:          header to be associated with the
                         image
        name:            the EXTNAME value; if this value
                         is None, then the name from the
                         input image header will be used;
                         if there is no name in the input
                         image header then the default name
                         'COMPRESSED_IMAGE' is used
        compressionType: compression algorithm 'RICE_1',
                         'PLIO_1', 'GZIP_1', 'HCOMPRESS_1'
        tileSize:        compression tile sizes default
                         treats each row of image as a tile
        hcompScale:      HCOMPRESS scale parameter
        hcompSmooth:     HCOMPRESS smooth parameter
        quantizeLevel:   floating point quantization level;
    """
```

- Added two new convenience functions. The setval function allows the setting of the value of a single header card in a fits file. The delval function allows the deletion of a single header card in a fits file.

- A modification was made to allow the reading of data from a fits file containing a Table HDU that has duplicate field names. It is normally a requirement that the field names in a Table HDU be unique. Prior to this change a ValueError was raised, when the data was accessed, to indicate that the HDU contained duplicate field names. Now, a warning is issued and the field names are made unique in the internal record array. This will not change the TTYPEn header card values. You will be able to get the data from all fields using the field name, including the first field containing the name that is duplicated. To access the data of the other fields with the duplicated names you will need to use the field number instead of the field name. (CNSHD737193)
- An enhancement was made to allow the reading of unsigned integer 16 values from an ImageHDU when the data is signed integer 16 and BZERO is equal to 32784 and BSCALE is equal to 1 (the standard way for scaling unsigned integer 16 data). A new optional keyword argument (uint16) was added to the open convenience function. Supplying a value of True for this argument will cause data of this type to be read in and scaled into an unsigned integer 16 array, instead of a float 32 array. If a HDU associated with a file that was opened with the uint16 option and containing unsigned integer 16 data is written to a file, the data will be reverse scaled into an integer 16 array and written out to the file and the BSCALE/BZERO header cards will be written with the values 1 and 32768 respectively. (CHSHD736064) Reference the following example:

```
>>> import pyfits
>>> hdul=pyfits.open('o4sp040b0_raw.fits',uint16=1)
>>> hdul[1].data
array([[1507, 1509, 1505, ..., 1498, 1500, 1487],
       [1508, 1507, 1509, ..., 1498, 1505, 1490],
       [1505, 1507, 1505, ..., 1499, 1504, 1491],
       ...,
       [1505, 1506, 1507, ..., 1497, 1502, 1487],
       [1507, 1507, 1504, ..., 1495, 1499, 1486],
       [1515, 1507, 1504, ..., 1492, 1498, 1487]], dtype=uint16)
>>> hdul.writeto('tmp.fits')
>>> hdul1=pyfits.open('tmp.fits',uint16=1)
>>> hdul1[1].data
array([[1507, 1509, 1505, ..., 1498, 1500, 1487],
       [1508, 1507, 1509, ..., 1498, 1505, 1490],
       [1505, 1507, 1505, ..., 1499, 1504, 1491],
       ...,
       [1505, 1506, 1507, ..., 1497, 1502, 1487],
       [1507, 1507, 1504, ..., 1495, 1499, 1486],
        [1515, 1507, 1504, ..., 1492, 1498, 1487]], dtype=uint16)
>>> hdul1=pyfits.open('tmp.fits')
>>> hdul1[1].data
array([[ 1507.,  1509.,  1505., ...,  1498.,  1500.,  1487.],
       [ 1508.,  1507.,  1509., ...,  1498.,  1505.,  1490.],
       [ 1505.,  1507.,  1505., ...,  1499.,  1504.,  1491.],
       ...,
       [ 1505.,  1506.,  1507., ...,  1497.,  1502.,  1487.],
       [ 1507.,  1507.,  1504., ...,  1495.,  1499.,  1486.],
       [ 1515.,  1507.,  1504., ...,  1492.,  1498.,  1487.]],
dtype=float32)
```

- Enhanced the message generated when a ValueError exception is raised when attempting to access a header card with an unparsable value. The message now includes the Card name.

The following bugs were fixed:

- Corrected a bug that occurs when appending a binary table HDU to a fits file. Data was not being byteswapped on little endian machines. (CNSHD737243)
- Corrected a bug that occurs when trying to write an ImageHDU that is missing the required PCOUNT card in the header. An UnboundLocalError exception complaining that the local variable 'insert_pos' was referenced before assignment was being raised in the method _ValidHDU.req_cards. The code was modified so that it would properly issue a more meaningful ValueError exception with a description of what required card is missing in the header.
- Eliminated a redundant warning message about the PCOUNT card when validating an ImageHDU header with a PCOUNT card that is missing or has a value other than 0.

1.4.1 (2008-11-04)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following enhancements were made:

- Enhanced the way import errors are reported to provide more information.

The following bugs were fixed:

- Corrected a bug that occurs when a card value is a string and contains a colon but is not a record-valued keyword card.
- Corrected a bug where pyfits fails to properly handle a record-valued

keyword card with values using exponential notation and trailing blanks.

1.4 (2008-07-07)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following enhancements were made:

- Added support for file objects and file like objects.
    - All convenience functions and class methods that take a file name will now also accept a file object or file like object. File like objects supported are StringIO and GzipFile objects. Other file like objects will work only if they implement all of the standard file object methods.
    - For the most part, file or file like objects may be either opened or closed at function call. An opened object must be opened with the proper mode depending on the function or method called. Whenever possible, if the object is opened before the method is called, it will remain open after the call. This will not be possible when writing a HDUList that has been resized or when writing to a GzipFile object regardless of whether it is resized. If the object is closed at the time of the function call, only the name from the object is used, not the object itself. The pyfits code will extract the file name used by the object and use that to create an underlying file object on which the function will be performed.
- Added support for record-valued keyword cards as introduced in the “FITS WCS proposal for representing a more general distortion model”.
    - Record-valued keyword cards are string-valued cards where the string is interpreted as a definition giving a record field name, and its floating point value. In a FITS header they have the following syntax:

      ```
      keyword= 'field-specifier: float'
      ```

      where keyword is a standard eight-character FITS keyword name, float is the standard FITS ASCII representation of a floating point number, and these are separated by a colon followed by a single blank.

      The grammar for field-specifier is:

      ```
      field-specifier:
          field
          field-specifier.field

      field:
      ```

```
    identifier
    identifier.index
```

where identifier is a sequence of letters (upper or lower case), underscores, and digits of which the first character must not be a digit, and index is a sequence of digits. No blank characters may occur in the field-specifier. The index is provided primarily for defining array elements though it need not be used for that purpose.

Multiple record-valued keywords of the same name but differing values may be present in a FITS header. The field-specifier may be viewed as part of the keyword name.

Some examples follow:

```
DP1     = 'NAXIS: 2'
DP1     = 'AXIS.1: 1'
DP1     = 'AXIS.2: 2'
DP1     = 'NAUX: 2'
DP1     = 'AUX.1.COEFF.0: 0'
DP1     = 'AUX.1.POWER.0: 1'
DP1     = 'AUX.1.COEFF.1: 0.00048828125'
DP1     = 'AUX.1.POWER.1: 1'
```

- As with standard header cards, the value of a record-valued keyword card can be accessed using either the index of the card in a HDU's header or via the keyword name. When accessing using the keyword name, the user may specify just the card keyword or the card keyword followed by a period followed by the field-specifier. Note that while the card keyword is case insensitive, the field-specifier is not. Thus, hdu['abc.def'], hdu['ABC.def'], or hdu['aBc.def'] are all equivalent but hdu['ABC.DEF'] is not.
- When accessed using the card index of the HDU's header the value returned will be the entire string value of the card. For example:

```
>>> print hdr[10]
NAXIS: 2
>>> print hdr[11]
AXIS.1: 1
```

- When accessed using the keyword name exclusive of the field-specifier, the entire string value of the header card with the lowest index having that keyword name will be returned. For example:

```
>>> print hdr['DP1']
NAXIS: 2
```

- When accessing using the keyword name and the field-specifier, the value returned will be the floating point value associated with the record-valued keyword card. For example:

```
>>> print hdr['DP1.NAXIS']
2.0
```

- Any attempt to access a non-existent record-valued keyword card value will cause an exception to be raised (IndexError exception for index access or KeyError for keyword name access).
- Updating the value of a record-valued keyword card can also be accomplished using either index or keyword name. For example:

```
>>> print hdr['DP1.NAXIS']
2.0
>>> hdr['DP1.NAXIS'] = 3.0
>>> print hdr['DP1.NAXIS']
3.0
```

- Adding a new record-valued keyword card to an existing header is accomplished using the Header.update() method just like any other card. For example:

```
>>> hdr.update('DP1', 'AXIS.3: 1', 'a comment',
after='DP1.AXIS.2')
```

- Deleting a record-valued keyword card from an existing header is accomplished using the standard list deletion syntax just like any other card. For example:

```
>>> del hdr['DP1.AXIS.1']
```

- In addition to accessing record-valued keyword cards individually using a card index or keyword name, cards can be accessed in groups using a set of special pattern matching keys. This access is made available via the standard list indexing operator providing a keyword name string that contains one or more of the special pattern matching keys. Instead of returning a value, a CardList object will be returned containing shared instances of the Cards in the header that match the given keyword specification.

  - There are three special pattern matching keys. The first key '*' will match any string of zero or more characters within the current level of the field-specifier. The second key '?' will match a single character. The third key '…' must appear at the end of the keyword name string and will match all keywords that match the preceding pattern down all levels of the field-specifier. All combinations of ?, *, and … are permitted (though … is only permitted at the end). Some examples follow:

    ```
    >>> cl=hdr['DP1.AXIS.*']
    >>> print cl
    DP1     = 'AXIS.1: 1'
    DP1     = 'AXIS.2: 2'
    >>> cl=hdr['DP1.*']
    >>> print cl
    DP1     = 'NAXIS: 2'
    DP1     = 'NAUX: 2'
    >>> cl=hdr['DP1.AUX...']
    >>> print cl
    DP1     = 'AUX.1.COEFF.0: 0'
    DP1     = 'AUX.1.POWER.0: 1'
    DP1     = 'AUX.1.COEFF.1: 0.00048828125'
    DP1     = 'AUX.1.POWER.1: 1'
    >>> cl=hdr['DP?.NAXIS']
    >>> print cl
    DP1     = 'NAXIS: 2'
    DP2     = 'NAXIS: 2'
    DP3     = 'NAXIS: 2'
    >>> cl=hdr['DP1.A*S.*']
    >>> print cl
    DP1     = 'AXIS.1: 1'
    DP1     = 'AXIS.2: 2'
    ```

  - The use of the special pattern matching keys for adding or updating header cards in an existing header is not allowed. However, the deletion of cards from the header using the special keys is allowed. For example:

    ```
    >>> del hdr['DP3.A*...']
    ```

- As noted above, accessing pyfits Header object using the special pattern matching keys will return a CardList object. This CardList object can itself be searched in order to further refine the list of Cards. For example:

  ```
  >>> cl=hdr['DP1...']
  >>> print cl
  DP1     = 'NAXIS: 2'
  DP1     = 'AXIS.1: 1'
  ```

```
DP1     = 'AXIS.2: 2'
DP1     = 'NAUX: 2'
DP1     = 'AUX.1.COEFF.1: 0.000488'
DP1     = 'AUX.2.COEFF.2: 0.00097656'
>>> cl1=cl['*.*AUX...']
>>> print cl1
DP1     = 'NAUX: 2'
DP1     = 'AUX.1.COEFF.1: 0.000488'
DP1     = 'AUX.2.COEFF.2: 0.00097656'
```

- The CardList keys() method will allow the retrieval of all of the key values in the CardList. For example:

```
>>> cl=hdr['DP1.AXIS.*']
>>> print cl
DP1     = 'AXIS.1: 1'
DP1     = 'AXIS.2: 2'
>>> cl.keys()
['DP1.AXIS.1', 'DP1.AXIS.2']
```

- The CardList values() method will allow the retrieval of all of the values in the CardList. For example:

```
>>> cl=hdr['DP1.AXIS.*']
>>> print cl
DP1     = 'AXIS.1: 1'
DP1     = 'AXIS.2: 2'
>>> cl.values()
[1.0, 2.0]
```

- Individual cards can be retrieved from the list using standard list indexing. For example:

```
>>> cl=hdr['DP1.AXIS.*']
>>> c=cl[0]
>>> print c
DP1     = 'AXIS.1: 1'
>>> c=cl['DP1.AXIS.2']
>>> print c
DP1     = 'AXIS.2: 2'
```

- Individual card values can be retrieved from the list using the value attribute of the card. For example:

```
>>> cl=hdr['DP1.AXIS.*']
>>> cl[0].value
```

```
1.0
```

- The cards in the CardList are shared instances of the cards in the source header. Therefore, modifying a card in the CardList also modifies it in the source header. However, making an addition or a deletion to the CardList will not affect the source header. For example:

```
>>> hdr['DP1.AXIS.1']
1.0
>>> cl=hdr['DP1.AXIS.*']
>>> cl[0].value = 4.0
>>> hdr['DP1.AXIS.1']
4.0
>>> del cl[0]
>>> print cl['DP1.AXIS.1']
Traceback (most recent call last):
...
KeyError: "Keyword 'DP1.AXIS.1' not found."
>>> hdr['DP1.AXIS.1']
4.0
```

- A FITS header consists of card images. In pyfits each card image is manifested by a Card object. A pyfits Header object contains a list of Card objects in the form of a CardList object. A record-valued keyword card image is represented in pyfits by a RecordValuedKeywordCard object. This object inherits from a Card object and has all of the methods and attributes of a Card object.
- A new RecordValuedKeywordCard object is created with the RecordValuedKeywordCard constructor: RecordValuedKeywordCard(key, value, comment). The key and value arguments may be specified in two ways. The key value may be given as the 8 character keyword only, in which case the value must be a character string containing the field-specifier, a colon followed by a space, followed by the actual value. The second option is to provide the key as a string containing the keyword and field-specifier, in which case the value must be the actual floating point value. For example:

```
>>> c1 = pyfits.RecordValuedKeywordCard('DP1', 'NAXIS: 2',
'Number of variables')
>>> c2 = pyfits.RecordValuedKeywordCard('DP1.AXIS.1', 1.0, 'Axis
number')
```

- RecordValuedKeywordCards have attributes .key, .field_specifier, .value, and .comment. Both .value and .comment can be changed but not .key or .field_specifier. The constructor will extract the field-specifier from the input key or value, whichever is appropriate. The .key attribute is the 8 character keyword.

- Just like standard Cards, a RecordValuedKeywordCard may be constructed from a string using the fromstring() method or verified using the verify() method. For example:

```
>>> c1 = pyfits.RecordValuedKeywordCard().fromstring(
         "DP1     = 'NAXIS: 2' / Number of independent
variables")
>>> c2 = pyfits.RecordValuedKeywordCard().fromstring(
         "DP1     = 'AXIS.1: X' / Axis number")
>>> print c1; print c2
DP1     = 'NAXIS: 2' / Number of independent variables
DP1     = 'AXIS.1: X' / Axis number
>>> c2.verify()
Output verification result:
Card image is not FITS standard (unparsable value string).
```

- A standard card that meets the criteria of a RecordValuedKeywordCard may be turned into a RecordValuedKeywordCard using the class method coerce. If the card object does not meet the required criteria then the original card object is just returned.

```
>>> c1 = pyfits.Card('DP1','AUX: 1','comment')
>>> c2 = pyfits.RecordValuedKeywordCard.coerce(c1)
>>> print type(c2)
<'pyfits.NP_pyfits.RecordValuedKeywordCard'>
```

- Two other card creation methods are also available as RecordVauedKeywordCard class methods. These are createCard() which will create the appropriate card object (Card or RecordValuedKeywordCard) given input key, value, and comment, and createCardFromString which will create the appropriate card object given an input string. These two methods are also available as convenience functions:

```
>>> c1 = pyfits.RecordValuedKeywordCard.createCard('DP1','AUX:
1','comment')
```

or

```
>>> c1 = pyfits.createCard('DP1','AUX: 1','comment')
>>> print type(c1)
<'pyfits.NP_pyfits.RecordValuedKeywordCard'>
```

```
>>> c1 = pyfits.RecordValuedKeywordCard.createCard('DP1','AUX 1','comment')
```

or

```
>>> c1 = pyfits.createCard('DP1','AUX 1','comment')
>>> print type(c1)
<'pyfits.NP_pyfits.Card'>
```

```
>>> c1 = pyfits.RecordValuedKeywordCard.createCardFromString \
         ("DP1 = 'AUX: 1.0' / comment")
```

or

```
>>> c1 = pyfits.createCardFromString("DP1     = 'AUX: 1.0' / comment")
>>> print type(c1)
<'pyfits.NP_pyfits.RecordValuedKeywordCard'>
```

The following bugs were fixed:

- Corrected a bug that occurs when writing a HDU out to a file. During the write, any Keyboard Interrupts are trapped so that the write completes before the interrupt is handled. Unfortunately, the Keyboard Interrupt was not properly reinstated after the write completed. This was fixed. (CNSHD711138)
- Corrected a bug when using ipython, where temporary files created with the tempFile.NamedTemporaryFile method are not automatically removed. This can happen for instance when opening a Gzipped fits file or when open a fits file over the internet. The files will now be removed. (CNSHD718307)
- Corrected a bug in the append convenience function’s call to the writeto convenience function. The classExtensions argument must be passed as a keyword argument.
- Corrected a bug that occurs when retrieving variable length character arrays from binary table HDUs (PA() format) and using slicing to obtain rows of data containing variable length arrays. The code issued a TypeError exception. The data can now be accessed with no exceptions. (CNSHD718749)
- Corrected a bug that occurs when retrieving data from a fits file opened in memory map mode when the file contains multiple image extensions or ASCII table or binary table HDUs. The code issued a TypeError exception. The data can now be accessed with no exceptions. (CNSHD707426)
- Corrected a bug that occurs when attempting to get a subset of data from a Binary Table HDU and then use the data to create a new Binary Table HDU object. A TypeError exception was raised. The data can now be subsetted

and used to create a new HDU. (CNSHD723761)
- Corrected a bug that occurs when attempting to scale an Image HDU back to its original data type using the _ImageBaseHDU.scale method. The code was not resetting the BITPIX header card back to the original data type. This has been corrected.
- Changed the code to issue a KeyError exception instead of a NameError exception when accessing a non-existent field in a table.

1.3 (2008-02-22)

Updates described in this release are only supported in the NUMPY version of pyfits.

The following enhancements were made:

- Provided support for a new extension to pyfits called *stpyfits*.
  - The *stpyfits* module is a wrapper around pyfits. It provides all of the features and functions of pyfits along with some STScI specific features. Currently, the only new feature supported by stpyfits is the ability to read and write fits files that contain image data quality extensions with constant data value arrays. See stpyfits [2] for more details on stpyfits.
- Added a new feature to allow trailing HDUs to be deleted from a fits file without actually reading the data from the file.
  - This supports a JWST requirement to delete a trailing HDU from a file whose primary Image HDU is too large to be read on a 32 bit machine.
- Updated pyfits to use the warnings module to issue warnings. All warnings will still be issued to stdout, exactly as they were before, however, you may now suppress warnings with the -Wignore command line option. For example, to run a script that will ignore warnings use the following command line syntax:

  ```
  python -Wignore yourscript.py
  ```

- Updated the open convenience function to allow the input of an already opened file object in place of a file name when opening a fits file.
- Updated the writeto convenience function to allow it to accept the output_verify option.
  - In this way, the user can use the argument output_verify='fix' to allow pyfits to correct any errors it encounters in the provided header before writing the data to the file.
- Updated the verification code to provide additional detail with a VerifyError exception.

- Added the capability to create a binary table HDU directly from a numpy.ndarray. This may be done using either the new_table convenience function or the BinTableHDU constructor.

The following performance improvements were made:

- Modified the import logic to dramatically decrease the time it takes to import pyfits.
- Modified the code to provide performance improvements when copying and examining header cards.

The following bugs were fixed:

- Corrected a bug that occurs when reading the data from a fits file that includes BZERO/BSCALE scaling. When the data is read in from the file, pyfits automatically scales the data using the BZERO/BSCALE values in the header. In the previous release, pyfits created a 32 bit floating point array to hold the scaled data. This could cause a problem when the value of BZERO is so large that the scaled value will not fit into the float 32. For this release, when the input data is 32 bit integer, a 64 bit floating point array is used for the scaled data.
- Corrected a bug that caused an exception to be raised when attempting to scale image data using the ImageHDU.scale method.
- Corrected a bug in the new_table convenience function that occurred when a binary table was created using a ColDefs object as input and supplying an nrows argument for a number of rows that is greater than the number of rows present in the input ColDefs object. The previous version of pyfits failed to allocate the necessary memory for the additional rows.
- Corrected a bug in the new_table convenience function that caused an exception to be thrown when creating an ASCII table.
- Corrected a bug in the new_table convenience function that will allow the input of a ColDefs object that was read from a file as a binary table with a data value equal to None.
- Corrected a bug in the construction of ASCII tables from Column objects that are created with noncontinuous start columns.
- Corrected bugs in a number of areas that would sometimes cause a failure to improperly raise an exception when an error occurred.
- Corrected a bug where attempting to open a non-existent fits file on a windows platform using a drive letter in the file specification caused a misleading IOError exception to be raised.

1.1 (2007-06-15)

- Modified to use either NUMPY or NUMARRAY.
- New file writing modes have been provided to allow streaming data to extensions without requiring the whole output extension image in memory. See documentation on StreamingHDU.
- Improvements to minimize byteswapping and memory usage by byteswapping in place.
- Now supports ‘:’ characters in filenames.
- Handles keyboard interrupts during long operations.
- Preserves the byte order of the input image arrays.

1.0.1 (2006-03-24)

The changes to PyFITS were primarily to improve the docstrings and to reclassify some public functions and variables as private. Readgeis and fitsdiff which were distributed with PyFITS in previous releases were moved to pytools. This release of PyFITS is v1.0.1. The next release of PyFITS will support both numarray and numpy (and will be available separately from stsci_python, as are all the python packages contained within stsci_python). An alpha release for PyFITS numpy support will be made around the time of this stsci_python release.

- Updated docstrings for public functions.
- Made some previously public functions private.

1.0 (2005-11-01)

Major Changes since v0.9.6:

- Added support for the HIERARCH convention
- Added support for iteration and slicing for HDU lists
- PyFITS now uses the standard setup.py installation script
- Add utility functions at the module level, they include:
  - getheader
  - getdata
  - getval
  - writeto
  - append
  - update
  - info

Minor changes since v0.9.6:

- Fix a bug to make single-column ASCII table work.
- Fix a bug so a new table constructed from an existing table with X-formatted columns will work.
- Fix a problem in verifying HDUList right after the open statement.
- Verify that elements in an HDUList, besides the first one, are ExtensionHDU.
- Add output verification in methods flush() and close().
- Modify the design of the open() function to remove the output_verify argument.
- Remove the groups argument in GroupsHDU's constructor.
- Redesign the column definition class to make its column components more accessible. Also to make it conducive for higher level functionalities, e.g. combining two column definitions.
- Replace the Boolean class with the Python Boolean type. The old TRUE/FALSE will still work.
- Convert classes to the new style.
- Better format when printing card or card list.
- Add the optional argument clobber to all writeto() functions and methods.
- If adding a blank card, will not use existing blank card's space.

PyFITS Version 1.0 REQUIRES Python 2.3 or later.

0.9.6 (2004-11-11)

Major changes since v0.9.3:

- Support for variable length array tables.
- Support for writing ASCII table extensions.
- Support for random groups, both reading and writing.

Some minor changes:

- Support for numbers with leading zeros in an ASCII table extension.
- Changed scaled columns' data type from Float32 to Float64 to preserve precision.
- Made Column constructor more flexible in accepting format specification.

0.9.3 (2004-07-02)

Changes since v0.9.0:

- Lazy instantiation of full Headers/Cards for all HDU's when the file is opened. At the open, only extracts vital info (e.g. NAXIS's) from the header parts. This

change will speed up the performance if the user only needs to access one extension in a multi-extension FITS file.
- Support the X format (bit flags) columns, both reading and writing, in a binary table. At the user interface, they are converted to Boolean arrays for easy manipulation. For example, if the column's TFORM is "11X", internally the data is stored in 2 bytes, but the user will see, at each row of this column, a Boolean array of 11 elements.
- Fix a bug such that when a table extension has no data, it will not try to scale the data when updating/writing the HDU list.

0.9 (2004-04-27)

Changes since v0.8.0:

- Rewriting of the Card class to separate the parsing and verification of header cards
- Restructure the keyword indexing scheme which speed up certain applications (update large number of new keywords and reading a header with larger numbers of cards) by a factor of 30 or more
- Change the default to be lenient FITS standard checking on input and strict FITS standard checking on output
- Support CONTINUE cards, both reading and writing
- Verification can now be performed at any of the HDUList, HDU, and Card levels
- Support (contiguous) subsection (attribute .section) of images to reduce memory usage for large images

0.8.0 (2003-08-19)

NOTE: This version will only work with numarray Version 0.6. In addition, earlier versions of PyFITS will not work with numarray 0.6. Therefore, both must be updated simultaneously.

Changes since 0.7.6:

- Compatible with numarray 0.6/records 2.0
- For binary tables, now it is possible to update the original array if a scaled field is updated.
- Support of complex columns
- Modify the __getitem__ method in FITS_rec. In order to make sure the scaled quantities are also viewing the same data as the original FITS_rec, all

fields need to be "touched" when __getitem__ is called.
- Add a new attribute mmobject for HDUList, and close the memmap object when close HDUList object. Earlier version does not close memmap object and can cause memory lockup.
- Enable 'update' as a legitimate memmap mode.
- Do not print message when closing an HDUList object which is not created from reading a FITS file. Such message is confusing.
- remove the internal attribute "closed" and related method (__getattr__ in HDUList). It is redundant.

0.7.6 (2002-11-22)

NOTE: This version will only work with numarray Version 0.4.

Changes since 0.7.5:

- Change x*=n to numarray.multiply(x, n, x) where n is a floating number, in order to make pyfits to work under Python 2.2. (2 occurrences)
- Modify the "update" method in the Header class to use the "fixed-format" card even if the card already exists. This is to avoid the mis-alignment as shown below:

  After running drizzle on ACS images it creates a CD matrix whose elements have very many digits, *e.g.*:

  > CD1_1 = 1.1187596304411E-05 / partial of first axis coordinate w.r.t. x CD1_2 = -8.502767249350019E-06 / partial of first axis coordinate w.r.t. y

  with pyfits, an "update" on these header items and write in new values which has fewer digits, *e.g.*:

  > CD1_1 = 1.0963011E-05 / partial of first axis coordinate w.r.t. x CD1_2 = -8.527229E-06 / partial of first axis coordinate w.r.t. y

- Change some internal variables to make their appearance more consistent:

  > old name new name
  >
  > __octalRegex _octalRegex __readblock() _readblock() __formatter() _formatter(). __value_RE _value_RE __numr _numr __comment_RE _comment_RE __keywd_RE _keywd_RE __number_RE _number_RE. tmpName() _tmpName() dimShape _dimShape ErrList _ErrList

- Move up the module description. Move the copyright statement to the bottom and assign to the variable __credits__.
- change the following line:

    self.__dict__ = input.__dict__

to

    self.__setstate__(input.__getstate__())

in order for pyfits to run under numarray 0.4.

- edit _readblock to add the (optional) firstblock argument and raise IOError if the first 8 characters in the first block is not ‘SIMPLE ‘ or ‘XTENSION’. Edit the function open to check for IOError to skip the last null filled block(s). Edit readHDU to add the firstblock argument.

0.7.5 (2002-08-16)

Changes since v0.7.3:

- Memory mapping now works for readonly mode, both for images and binary tables.

  Usage: pyfits.open(‘filename’, memmap=1)
- Edit the field method in FITS_rec class to make the column scaling for numbers use less temporary memory. (does not work under 2.2, due to Python “bug” of array *=)
- Delete bscale/bzero in the ImageBaseHDU constructor.
- Update bitpix in BaseImageHDU.__getattr__ after deleting bscale/bzero. (bug fix)
- In BaseImageHDU.__getattr__ point self.data to raw_data if float and if not memmap. (bug fix).
- Change the function get_tbdata() to private: _get_tbdata().

0.7.3 (2002-07-12)

Changes since v0.7.2:

- It will scale all integer image data to Float32, if BSCALE/BZERO != 1/0. It will also expunge the BSCALE/BZERO keywords.
- Add the scale() method for ImageBaseHDU, so data can be scaled just before being written to the file. It has the following arguments:

  type: destination data type (string), e.g. Int32, Float32, UInt8, etc.

option: scaling scheme. if 'old', use the old BSCALE/BZERO values. if 'minmax', use the data range to fit into the full range of specified integer type. Float destination data type will not be scaled for this option.

bscale/bzero: user specifiable BSCALE/BZERO values. They overwrite the "option".

- Deal with data area resizing in 'update' mode.
- Make the data scaling (both input and output) faster and use less memory.
- Bug fix to make column name change takes effect for field.
- Bug fix to avoid exception if the key is not present in the header already. This affects (fixes) add_history(), add_comment(), and add_blank().
- Bug fix in __getattr__() in Card class. The change made in 0.7.2 to rstrip the comment must be string type to avoid exception.

0.7.2.1 (2002-06-25)

A couple of bugs were addressed in this version.

- Fix a bug in _add_commentary(). Due to a change in index_of() during version 0.6.5.5, _add_commentary needs to be modified to avoid exception if the key is not present in the header already. This affects (fixes) add_history(), add_comment(), and add_blank().
- Fix a bug in __getattr__() in Card class. The change made in 0.7.2 to rstrip the comment must be string type to avoid exception.

0.7.2 (2002-06-19)

The two major improvements from Version 0.6.2 are:

- support reading tables with "scaled" columns (e.g. tscal/tzero, Boolean, and ASCII tables)
- a prototype output verification.

This version of PyFITS requires numarray version 0.3.4.

Other changes include:

- Implement the new HDU hierarchy proposed earlier this year. This in turn reduces some of the redundant methods common to several HDU classes.
- Add 3 new methods to the Header class: add_history, add_comment, and add_blank.

- The table attributes _columns are now .columns and the attributes in ColDefs are now all without the underscores. So, a user can get a list of column names by: hdu.columns.names.
- The “fill” argument in the new_table method now has a new meaning:<br> If set to true (=1), it will fill the entire new table with zeros/blanks. Otherwise (=0), just the extra rows/cells are filled with zeros/blanks. Fill values other than zero/blank are now not possible.
- Add the argument output_verify to the open method and writeto method. Not in the flush or close methods yet, due to possible complication.
- A new copy method for tables, the copy is totally independent from the table it copies from.
- The tostring() call in writeHDUdata takes up extra space to store the string object. Use tofile() instead, to save space.
- Make changes from _byteswap to _byteorder, following corresponding changes in numarray and recarray.
- Insert(update) EXTEND in PrimaryHDU only when header is None.
- Strip the trailing blanks for the comment value of a card.
- Add seek(0) right after the __buildin__.open(0), because for the ‘ab+’ mode, the pointer is at the end after open in Linux, but it is at the beginning in Solaris.
- Add checking of data against header, update header keywords (NAXIS’s, BITPIX) when they don’t agree with the data.
- change version to __version__.

There are also many other minor internal bug fixes and technical changes.

0.6.2 (2002-02-12)

This version requires numarray version 0.2.

Things not yet supported but are part of future development:

- Verification and/or correction of FITS objects being written to disk so that they are legal FITS. This is being added now and should be available in about a month. Currently, one may construct FITS headers that are inconsistent with the data and write such FITS objects to disk. Future versions will provide options to either a) correct discrepancies and warn, b) correct discrepancies silently, c) throw a Python exception, or d) write illegal FITS (for test purposes!).
- Support for ascii tables or random groups format. Support for ASCII tables will be done soon (~1 month). When random group support is added is uncertain.
- Support for memory mapping FITS data (to reduce memory demands). We

expect to provide this capability in about 3 months.
- Support for columns in binary tables having scaled values (e.g. BSCALE or BZERO) or boolean values. Currently booleans are stored as Int8 arrays and users must explicitly convert them into a boolean array. Likewise, scaled columns must be copied with scaling and offset by testing for those attributes explicitly. Future versions will produce such copies automatically.
- Support for tables with TNULL values. This awaits an enhancement to numarray to support mask arrays (planned). (At least a couple of months off).

Performance Tips

It is possible to set the data array for `PrimaryHDU` and `ImageHDU` to a dask array. If this is written to disk, the dask array will be computed as it is being written, which will avoid using excessive memory:

```
>>> import dask.array as da
>>> array = da.random.random((1000, 1000))
>>> from astropy.io import fits
>>> hdu = fits.PrimaryHDU(data=array)
>>> hdu.writeto('test_dask.fits')
```

Reference/API

A package for reading and writing FITS files and manipulating their contents.

A module for reading and writing Flexible Image Transport System (FITS) files. This file format was endorsed by the International Astronomical Union in 1999 and mandated by NASA as the standard format for storing high energy astrophysics data. For details of the FITS standard, see the NASA/Science Office of Standards and Technology publication, NOST 100-2.0.

File Handling and Convenience Functions

open()

`astropy.io.fits.` **`open`**(*name*, *mode='readonly'*, *memmap=None*, *save_backup=False*, *cache=True*, *lazy_load_hdus=None*, ***kwargs*)

Factory function to open a FITS file and return an `HDUList` object.

Parameters: **name** : *str, file-like or* **`pathlib.Path`**

File to be opened.

mode : *str, optional*

Open mode, 'readonly', 'update', 'append', 'denywrite', or 'ostream'. Default is 'readonly'.

If `name` is a file object that is already opened, `mode` must match the mode the file was opened with, readonly (rb), update (rb+), append (ab+), ostream (w), denywrite (rb)).

memmap : *bool, optional*

Is memory mapping to be used? This value is obtained from the configuration item `astropy.io.fits.Conf.use_memmap`. Default is **`True`**.

save_backup : *bool, optional*

If the file was opened in update or append mode, this ensures that a backup of the original file is saved before any changes are flushed. The backup has the same name as the original file with ".bak" appended. If "file.bak" already exists then "file.bak.1" is used, and so on. Default is **`False`**.

cache : *bool, optional*

If the file name is a URL, **`download_file`** is used to open the file. This specifies whether or not to save the file locally in Astropy's download cache. Default is **`True`**.

lazy_load_hdus : *bool, optional*

To avoid reading all the HDUs and headers in a FITS file immediately upon opening. This is an optimization especially useful for large files, as FITS has no way of determining the number and offsets of all the HDUs in a file without scanning through the file and reading all the headers. Default is **`True`**.

To disable lazy loading and read all HDUs immediately (the old behavior) use `lazy_load_hdus=False`. This can lead to fewer surprises–for example with lazy loading enabled, `len(hdul)` can be slow, as it means the entire FITS file needs to be read in order to determine the number of HDUs. `lazy_load_hdus=False` ensures that all HDUs have already been loaded after the file has been opened.

New in version 1.3.

uint : *bool, optional*

Interpret signed integer data where `BZERO` is the central value and `BSCALE == 1` as unsigned integer data. For example,

Returns:

hdulist : *an* ***HDUList*** *object*

HDUList containing all of the header data units in the file.

writeto()

`astropy.io.fits.` **`writeto`**(*filename*, *data*, *header=None*, *output_verify='exception'*, *overwrite=False*, *checksum=False*)

Create a new FITS file using the supplied data/header.

Parameters:

filename : *file path, file object, or file like object*

File to write to. If opened, must be opened in a writeable binary mode such as 'wb' or 'ab+'.

data : *array, record array, or groups data object*

data to write to the new file

header : ***Header*** *object, optional*

the header associated with `data`. If **None**, a header of the appropriate type is created for the supplied data. This argument is optional.

output_verify : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `+warn`, or `+exception" (e.g. ``"fix+warn"`). See Verification Options for more info.

overwrite : *bool, optional*

If `True`, overwrite the output file if it exists. Raises an `OSError` if `False` and the output file exists. Default is `False`.

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

checksum : *bool, optional*

If **True**, adds both `DATASUM` and `CHECKSUM` cards to the headers of all HDU's written to the file.

info()

`astropy.io.fits.` **`info`** (*filename*, *output=None*, ***kwargs*)

Print the summary information on a FITS file.

This includes the name, type, length of header, data shape and type for each extension.

Parameters: **filename** : *file path, file object, or file like object*

FITS file to obtain info from. If opened, mode must be one of the following: rb, rb+, or ab+ (i.e. the file must be readable).

output : *file, bool, optional*

A file-like object to write the output to. If `False`, does not output to a file and instead returns a list of tuples representing the HDU info. Writes to `sys.stdout` by default.

kwargs

Any additional keyword arguments to be passed to **`astropy.io.fits.open`**. *Note:* This function sets `ignore_missing_end=True` by default.

printdiff()

`astropy.io.fits.` **`printdiff`** (*inputa*, *inputb*, **args*, ***kwargs*)

Compare two parts of a FITS file, including entire FITS files, FITS **`HDUList`** objects and FITS `HDU` objects.

Parameters: **inputa** : *str, `HDUList` object, or `HDU` object*

The filename of a FITS file, `HDUList`, or `HDU` object to compare to `inputb`.

inputb : *str, `HDUList` object, or `HDU` object*

The filename of a FITS file, `HDUList`, or `HDU` object to compare to `inputa`.

ext, extname, extver

Additional positional arguments are for extension specification if your inputs are string filenames (will not work if `inputa` and `inputb` are `HDU` objects or `HDUList` objects). They are flexible and are best illustrated by examples. In addition to using these arguments positionally you can directly call the keyword parameters `ext`, `extname`.
By extension number:

```
printdiff('inA.fits', 'inB.fits', 0)      # the primary
HDU
printdiff('inA.fits', 'inB.fits', 2)      # the second
extension
printdiff('inA.fits', 'inB.fits', ext=2)  # the second
extension
```

By name, i.e., `EXTNAME` value (if unique). `EXTNAME` values are not case sensitive:

> printdiff('inA.fits', 'inB.fits', 'sci') printdiff('inA.fits', 'inB.fits', extname='sci') # equivalent

By combination of `EXTNAME` and `EXTVER` as separate arguments or as a tuple:

```
printdiff('inA.fits', 'inB.fits', 'sci', 2)    #
EXTNAME='SCI'
                                               # &
EXTVER=2
printdiff('inA.fits', 'inB.fits', extname='sci',
extver=2)
                                               #
equivalent
printdiff('inA.fits', 'inB.fits', ('sci', 2))  #
equivalent
```

Ambiguous or conflicting specifications will raise an exception:

Notes

The primary use for the **`printdiff`** function is to allow quick print out of a FITS difference report and will write to `sys.stdout`. To save the diff report to a file please use **`FITSDiff`** directly.

append()

`astropy.io.fits.` **`append`**(*filename*, *data*, *header=None*, *checksum=False*, *verify=True*, ***kwargs*)

Append the header/data to FITS file if filename exists, create if not.

If only `data` is supplied, a minimal header is created.

Parameters: **filename** : *file path, file object, or file like object*

File to write to. If opened, must be opened for update (rb+) unless it is a new file, then it must be opened for append (ab+). A file or `GzipFile` object opened for update will be closed after return.

data : *array, table, or group data object*

the new data used for appending

header : *`Header` object, optional*

The header associated with `data`. If `None`, an appropriate header will be created for the data object supplied.

checksum : *bool, optional*

When `True` adds both `DATASUM` and `CHECKSUM` cards to the header of the HDU when written to the file.

verify : *bool, optional*

When `True`, the existing FITS file will be read in to verify it for correctness before appending. When `False`, content is simply appended to the end of the file. Setting `verify` to `False` can be much faster.

kwargs

Additional arguments are passed to:

- `writeto` if the file does not exist or is empty. In this case `output_verify` is the only possible argument.
- `open` if `verify` is True or if `filename` is a file object.
- Otherwise no additional arguments can be used.

`update()`

`astropy.io.fits.` **`update`**(*filename*, *data*, **args*, ***kwargs*)

Update the specified extension with the input data/header.

Parameters: **filename** : *file path, file object, or file like object*

File to update. If opened, mode must be update (rb+). An opened file object or `GzipFile` object will be closed upon return.

data : *array, table, or group data object*

the new data used for updating

header : *`Header` object, optional*

The header associated with `data`. If `None`, an appropriate header will be created for the data object supplied.

ext, extname, extver

The rest of the arguments are flexible: the 3rd argument can be the header associated with the data. If the 3rd argument is not a `Header`, it (and other positional arguments) are assumed to be the extension specification(s). Header and extension specs can also be keyword arguments. For example:

```
update(file, dat, hdr, 'sci')  # update the 'sci'
extension
update(file, dat, 3)  # update the 3rd extension
update(file, dat, hdr, 3)  # update the 3rd extension
update(file, dat, 'sci', 2)  # update the 2nd SCI
extension
update(file, dat, 3, header=hdr)  # update the 3rd
extension
update(file, dat, header=hdr, ext=5)  # update the 5th
extension
```

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`.

getdata()

`astropy.io.fits.` **`getdata`**(*filename*, **args*, *header=None*, *lower=None*, *upper=None*, *view=None*, ***kwargs*)

Get the data from an extension of a FITS file (and optionally the header).

Parameters: **filename** : *file path, file object, or file like object*

File to get data from. If opened, mode must be one of the following rb, rb+, or ab+.

ext

The rest of the arguments are for extension specification. They are flexible and are best illustrated by examples.
No extra arguments implies the primary header:

```
getdata('in.fits')
```

By extension number:

```
getdata('in.fits', 0)      # the primary header
getdata('in.fits', 2)      # the second extension
getdata('in.fits', ext=2)  # the second extension
```

By name, i.e., `EXTNAME` value (if unique):

```
getdata('in.fits', 'sci')
getdata('in.fits', extname='sci')  # equivalent
```

Note `EXTNAME` values are not case sensitive
By combination of `EXTNAME` and EXTVER`` as separate arguments or as a tuple:

```
getdata('in.fits', 'sci', 2)  # EXTNAME='SCI' &
EXTVER=2
getdata('in.fits', extname='sci', extver=2)  #
equivalent
getdata('in.fits', ('sci', 2))  # equivalent
```

Ambiguous or conflicting specifications will raise an exception:

```
getdata('in.fits', ext=('sci',1), extname='err',
extver=2)
```

header : *bool, optional*

If **`True`**, return the data and the header of the specified HDU as a tuple.

lower, upper : *bool, optional*

If `lower` or `upper` are **`True`**, the field names in the returned data object will be converted to lower or upper case, respectively.

Returns: **array** : *array, record array or groups data object*

Type depends on the type of the extension being referenced. If the optional keyword `header` is set to `True`, this function will return a (`data` , `header`) tuple.

getheader()

`astropy.io.fits.` **`getheader`** (*filename*, **args*, ***kwargs*)

Get the header from an extension of a FITS file.

Parameters: **filename** : *file path, file object, or file like object*

File to get header from. If an opened file object, its mode must be one of the following rb, rb+, or ab+).

ext, extname, extver

The rest of the arguments are for extension specification. See the `getdata` documentation for explanations/examples.

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`.

Returns: **header** : *`Header` object*

getval()

`astropy.io.fits.` **`getval`** (*filename*, *keyword*, **args*, ***kwargs*)

Get a keyword’s value from a header in a FITS file.

Parameters: **filename** : *file path, file object, or file like object*

Name of the FITS file, or file object (if opened, mode must be one of the following rb, rb+, or ab+).

keyword : *str*

Keyword name

ext, extname, extver

The rest of the arguments are for extension specification. See `getdata` for explanations/examples.

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`. *Note:* This function automatically specifies `do_not_scale_image_data = True` when opening the file so that values can be retrieved from the unmodified header.

Returns: **keyword value** : *str, int, or float*

setval()

`astropy.io.fits.` **`setval`**(*filename*, *keyword*, **args*, *value=None*, *comment=None*, *before=None*, *after=None*, *savecomment=False*, ***kwargs*)

Set a keyword's value from a header in a FITS file.

If the keyword already exists, it's value/comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

When updating more than one keyword in a file, this convenience function is a much less efficient approach compared with opening the file for update, modifying the header, and closing the file.

Parameters: **filename** : *file path, file object, or file like object*

Name of the FITS file, or file object If opened, mode must be update (rb+). An opened file object or `GzipFile` object will be closed upon return.

keyword : *str*

Keyword name

value : *str, int, float, optional*

Keyword value (default: `None`, meaning don't modify)

comment : *str, optional*

Keyword comment, (default: `None`, meaning don't modify)

before : *str, int, optional*

Name of the keyword, or index of the card before which the new card will be placed. The argument `before` takes precedence over `after` if both are specified (default: `None`).

after : *str, int, optional*

Name of the keyword, or index of the card after which the new card will be placed. (default: `None`).

savecomment : *bool, optional*

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved (default: `False`).

ext, extname, extver

The rest of the arguments are for extension specification. See `getdata` for explanations/examples.

kwargs

Any additional keyword arguments to be passed to `astropy.io.fits.open`. *Note:* This function automatically specifies `do_not_scale_image_data = True` when opening the file so that values can be retrieved from the unmodified header.

`delval()`

`astropy.io.fits.` **`delval`** (*filename*, *keyword*, **args*, ***kwargs*)

Delete all instances of keyword from a header in a FITS file.

Parameters: **filename** : *file path, file object, or file like object*

Name of the FITS file, or file object If opened, mode must be update (rb+). An opened file object or **`GzipFile`** object will be closed upon return.

keyword : *str, int*

Keyword name or index

ext, extname, extver

The rest of the arguments are for extension specification. See **`getdata`** for explanations/examples.

kwargs

Any additional keyword arguments to be passed to **`astropy.io.fits.open`**. *Note:* This function automatically specifies `do_not_scale_image_data = True` when opening the file so that values can be retrieved from the unmodified header.

HDU Lists

HDUList

class `astropy.io.fits.` **`HDUList`** (*hdus=[]*, *file=None*)

Bases: **`list`**, **`astropy.io.fits.verify._Verify`**

HDU list class. This is the top-level FITS object. When a FITS file is opened, a **`HDUList`** object is returned.

Construct a **`HDUList`** object.

Parameters: **hdus** : *sequence of HDU objects or single HDU, optional*

The HDU object(s) to comprise the `HDUList`. Should be instances of HDU classes like `ImageHDU` or `BinTableHDU`.

file : *file object, bytes, optional*

The opened physical file associated with the `HDUList` or a bytes object containing the contents of the FITS file.

`append` (*hdu*)

Append a new HDU to the `HDUList`.

Parameters: **hdu** : *HDU object*

HDU to add to the `HDUList`.

`close` (*output_verify='exception'*, *verbose=False*, *closed=True*)

Close the associated FITS file and memmap object, if any.

Parameters: **output_verify** : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `+warn`, or `+exception" (e.g. ``"fix+warn"`). See Verification Options for more info.

verbose : *bool*

When `True`, print out verbose messages.

closed : *bool*

When `True`, close the underlying file object.

`copy` ()

Return a shallow copy of an HDUList.

Returns: **copy** : *`HDUList`*

A shallow copy of this `HDUList` object.

`fileinfo` (*index*)

Returns a dictionary detailing information about the locations of the indexed HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the `HDUList`.

Parameters: **index** : *int*

Index of HDU for which info is to be returned.

Returns: **fileinfo** : *dict or None*

The dictionary details information about the locations of the indexed HDU within an associated file. Returns `None` when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
filename	Name of associated file object
filemode	Mode in which the file was opened (readonly, update, append, denywrite, ostream)
resized	Flag that when `True` indicates that the data has been resized since the last read/write so the returned values may not be valid.
hdrLoc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

`filename` ()

Return the file name associated with the HDUList object if one exists. Otherwise returns None.

Returns: **filename** : *a string containing the file name associated with the*

HDUList object if an association exists. Otherwise returns None.

`flush` (*output_verify='fix'*, *verbose=False*)

Force a write of the `HDUList` back to the file (for append and update modes only).

Parameters: **output_verify** : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `+warn`, or `+exception" (e.g. ``"fix+warn"`). See Verification Options for more info.

verbose : *bool*

When `True`, print verbose messages

classmethod `fromfile` (*fileobj*, *mode=None*, *memmap=None*, *save_backup=False*, *cache=True*, *lazy_load_hdus=True*, ***kwargs*)

Creates an `HDUList` instance from a file-like object.

The actual implementation of `fitsopen()`, and generally shouldn't be used directly. Use **`open()`** instead (and see its documentation for details of the parameters accepted by this method).

classmethod **`fromstring`**(*data*, ***kwargs*)

Creates an **`HDUList`** instance from a string or other in-memory data buffer containing an entire FITS file. Similar to **`HDUList.fromfile()`**, but does not accept the mode or memmap arguments, as they are only relevant to reading from a file on disk.

This is useful for interfacing with other libraries such as CFITSIO, and may also be useful for streaming applications.

Parameters: **data** : *str, buffer, memoryview, etc.*

A string or other memory buffer containing an entire FITS file. It should be noted that if that memory is read-only (such as a Python string) the returned **`HDUList`**'s data portions will also be read-only.

kwargs : *dict*

Optional keyword arguments. See **`astropy.io.fits.open()`** for details.

Returns: **hdul** : *HDUList*

An **`HDUList`** object representing the in-memory FITS file.

`index_of`(*key*)

Get the index of an HDU from the **`HDUList`**.

Parameters: **key** : *int, str, tuple of (string, int) or an HDU object*

The key identifying the HDU. If `key` is a tuple, it is of the form `(name, ver)` where `ver` is an `EXTVER` value that must match the HDU being searched for.
If the key is ambiguous (e.g. there are multiple 'SCI' extensions) the first match is returned. For a more precise match use the `(name, ver)` pair.
If even the `(name, ver)` pair is ambiguous (it shouldn't be but it's not impossible) the numeric index must be used to index the duplicate HDU.
When `key` is an HDU object, this function returns the index of that HDU object in the `HDUList`.

Returns: **index** : *int*

The index of the HDU in the **`HDUList`**.

Raises: ValueError

If `key` is an HDU object and it is not found in the `HDUList`.

KeyError

If an HDU specified by the `key` that is an extension number, extension name, or a tuple of extension name and version is not found in the `HDUList`.

`info` (*output=None*)

Summarize the info of the HDUs in this **`HDUList`**.

Note that this function prints its results to the console—it does not return a value.

Parameters: **output** : *file, bool, optional*

A file-like object to write the output to. If **`False`**, does not output to a file and instead returns a list of tuples representing the HDU info. Writes to `sys.stdout` by default.

`insert` (*index*, *hdu*)

Insert an HDU into the **`HDUList`** at the given `index`.

Parameters: **index** : *int*

Index before which to insert the new HDU.

hdu : *HDU object*

The HDU object to insert

`pop` (*index=- 1*)

Remove an item from the list and return it.

Parameters: **index** : *int, str, tuple of (string, int), optional*

An integer value of `index` indicates the position from which `pop()` removes and returns an HDU. A string value or a tuple of `(string, int)` functions as a key for identifying the HDU to be removed and returned. If `key` is a tuple, it is of the form `(key, ver)` where `ver` is an `EXTVER` value that must match the HDU being searched for.
If the key is ambiguous (e.g. there are multiple 'SCI' extensions) the first match is returned. For a more precise match use the `(name, ver)` pair.
If even the `(name, ver)` pair is ambiguous the numeric index must be used to index the duplicate HDU.

Returns: **hdu** : *HDU object*

The HDU object at position indicated by `index` or having name and version specified by `index`.

`readall` ()

Read data of all HDUs into memory.

`update_extend` ()

Make sure that if the primary header needs the keyword `EXTEND` that it has it and it is correct.

`writeto` (*fileobj*, *output_verify='exception'*, *overwrite=False*, *checksum=False*)

Write the `HDUList` to a new file.

Parameters: **fileobj** : *str, file-like or `pathlib.Path`*

File to write to. If a file object, must be opened in a writeable mode.

output_verify : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `+warn`, or `+exception" (e.g. ``"fix+warn"`). See Verification Options for more info.

overwrite : *bool, optional*

If `True`, overwrite the output file if it exists. Raises an `OSError` if `False` and the output file exists. Default is `False`.

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

checksum : *bool*

When `True` adds both `DATASUM` and `CHECKSUM` cards to the headers of all HDU's written to the file.

Header Data Units

The `ImageHDU` and `CompImageHDU` classes are discussed in the section on Images.

The `TableHDU` and `BinTableHDU` classes are discussed in the section on Tables.

PrimaryHDU

class `astropy.io.fits.` **`PrimaryHDU`**(*data=None*, *header=None*, *do_not_scale_image_data=False*, *ignore_blank=False*, *uint=True*, *scale_back=None*)

Bases: **`astropy.io.fits.hdu.image._ImageBaseHDU`**

FITS primary HDU class.

Construct a primary HDU.

Parameters:

data : *array or DELAYED, optional*

The data in the HDU.

header : *`Header`, optional*

The header to be used (as a template). If `header` is `None`, a minimal header will be provided.

do_not_scale_image_data : *bool, optional*

If `True`, image data is not scaled using BSCALE/BZERO values when read. (default: False)

ignore_blank : *bool, optional*

If `True`, the BLANK header keyword will be ignored if present. Otherwise, pixels equal to this value will be replaced with NaNs. (default: False)

uint : *bool, optional*

Interpret signed integer data where `BZERO` is the central value and `BSCALE == 1` as unsigned integer data. For example, `int16` data with `BZERO = 32768` and `BSCALE = 1` would be treated as `uint16` data. (default: True)

scale_back : *bool, optional*

If `True`, when saving changes to a file that contained scaled image data, restore the data to the original type and reapply the original BSCALE/BZERO values. This could lead to loss of accuracy if scaling back to integer values after performing floating point operations on the data. Pseudo-unsigned integers are automatically rescaled unless scale_back is explicitly set to `False`. (default: None)

`add_checksum`(*when=None*, *override_datasum=False*, *checksum_keyword='CHECKSUM'*, *datasum_keyword='DATASUM'*)

Add the `CHECKSUM` and `DATASUM` cards to this HDU with the values set to the checksum calculated for the HDU and the data respectively. The addition of the `DATASUM` card may be overridden.

Parameters: **when** : *str, optional*

comment string for the cards; by default the comments will represent the time when the checksum was calculated

override_datasum : *bool, optional*

add the `CHECKSUM` card only

checksum_keyword : *str, optional*

The name of the header keyword to store the checksum value in; this is typically 'CHECKSUM' per convention, but there exist use cases in which a different keyword should be used

datasum_keyword : *str, optional*

See `checksum_keyword`

Notes

For testing purposes, first call **`add_datasum`** with a `when` argument, then call **`add_checksum`** with a `when` argument and `override_datasum` set to **`True`**. This will provide consistent comments for both cards and enable the generation of a `CHECKSUM` card with a consistent value.

`add_datasum`(*when=None, datasum_keyword='DATASUM'*)

Add the `DATASUM` card to this HDU with the value set to the checksum calculated for the data.

Parameters: **when** : *str, optional*

Comment string for the card that by default represents the time when the checksum was calculated

datasum_keyword : *str, optional*

The name of the header keyword to store the datasum value in; this is typically 'DATASUM' per convention, but there exist use cases in which a different keyword should be used

Returns: **checksum** : *int*

The calculated datasum

Notes

For testing purposes, provide a `when` argument to enable the comment value in the card to remain consistent. This will enable the generation of a `CHECKSUM` card with a consistent value.

`copy`()

Make a copy of the HDU, both header and data are copied.

property `data`

Image/array data as a `ndarray`.

Please remember that the order of axes on an Numpy array are opposite of the order specified in the FITS file. For example for a 2D image the "rows" or y-axis are the first dimension, and the "columns" or x-axis are the second dimension.

If the data is scaled using the BZERO and BSCALE parameters, this attribute returns the data scaled to its physical values unless the file was opened with `do_not_scale_image_data=True`.

`filebytes` ()

Calculates and returns the number of bytes that this HDU will write to a file.

`fileinfo` ()

Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the `HDUList`.

Returns: dict or None

The dictionary details information about the locations of this HDU within an associated file. Returns `None` when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
filemode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
hdrLoc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

classmethod `fromstring` (*data*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Creates a new HDU object of the appropriate type from a string containing the HDU's entire header and, optionally, its data.

Note: When creating a new HDU from a string without a backing file object, the data of that HDU may be read-only. It depends on whether the underlying string was an immutable Python str/bytes object, or some kind of read-write memory buffer such as a `memoryview`.

Parameters: **data** : *str, bytearray, memoryview, ndarray*

A byte string containing the HDU's header and data.

checksum : *bool, optional*

Check the HDU's checksum and/or datasum.

ignore_missing_end : *bool, optional*

Ignore a missing end card in the header data. Note that without the end card the end of the header may be ambiguous and resulted in a corrupt HDU. In this case the assumption is that the first 2880 block that does not begin with valid FITS header data is the beginning of the data.

kwargs : *optional*

May consist of additional keyword arguments specific to an HDU type–these correspond to keywords recognized by the constructors of different HDU classes such as `PrimaryHDU`, `ImageHDU`, or `BinTableHDU`. Any unrecognized keyword arguments are simply ignored.

classmethod `match_header`(*header*)

_ImageBaseHDU is sort of an abstract class for HDUs containing image data (as opposed to table data) and should never be used directly.

classmethod `readfrom`(*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with `open()` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto()`.

Parameters: **fileobj** : *file object or file-like object*

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : *bool*

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : *bool*

Do not issue an exception when opening a file that is missing an `END` card in the last header.

`req_cards`(*keyword*, *pos*, *test*, *fix_value*, *option*, *errlist*)

Check the existence, location, and value of a required `Card`.

Parameters: **keyword** : *str*

The keyword to validate

pos : *int, callable*

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument–the actual position of the keyword–and return **`True`** or **`False`**. This can be used for custom evaluation. For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : *callable*

This should be a callable (generally a function) that is passed the value of the given keyword and returns **`True`** or **`False`**. This can be used to validate the value associated with the given keyword.

fix_value : *str, int, float, complex, bool, None*

A valid value for a FITS keyword to to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If **`None`**, there is no replacement value and the keyword is unfixable.

option : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `+warn`, or `+exception" (e.g. ``"fix+warn"`). See Verification Options for more info.

errlist : *list*

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to **`req_cards`**.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

run_option (*option='warn', err_text='', fix_text='Fixed.', fix=None, fixable=True*)

Execute the verification with selected option.

scale (*type=None, option='old', bscale=None, bzero=None*)

Scale image data by using `BSCALE` / `BZERO`.

Call to this method will scale **data** and update the keywords of `BSCALE` and `BZERO` in the HDU's header. This method should only be used right before writing to the output file, as the data will be scaled and is therefore not very usable after the call.

Parameters: **type** : *str, optional*

destination data type, use a string representing a numpy dtype name, (e.g. `'uint8'`, `'int16'`, `'float32'` etc.). If is **None**, use the current data type.

option : *str, optional*

How to scale the data: `"old"` uses the original `BSCALE` and `BZERO` values from when the data was read/created (defaulting to 1 and 0 if they don't exist). For integer data only, `"minmax"` uses the minimum and maximum of the data to scale. User-specified `bscale`/`bzero` values always take precedence.

bscale, bzero : *int, optional*

User-specified `BSCALE` and `BZERO` values

property **section**

Access a section of the image array without loading the entire array into memory. The **Section** object returned by this attribute is not meant to be used directly by itself. Rather, slices of the section return the appropriate slice of the data, and loads *only* that section into memory.

Sections are mostly obsoleted by memmap support, but should still be used to deal with very large scaled images. See the Data Sections section of the Astropy documentation for more details.

property **shape**

Shape of the image array–should be equivalent to `self.data.shape`.

property **size**

Size (in bytes) of the data portion of the HDU.

update_header ()

Update the header keywords to agree with the data.

`verify` (*option='warn'*)

Verify all values in the instance.

Parameters: **option** : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `"+warn"`, or `"+exception"` (e.g. `"fix+warn"`). See Verification Options for more info.

`verify_checksum` ()

Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU CHECKSUM.

Returns: **valid** : *int*

- 0 - failure
- 1 - success
- 2 - no `CHECKSUM` keyword present

`verify_datasum` ()

Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

Returns: **valid** : *int*

- 0 - failure
- 1 - success
- 2 - no `DATASUM` keyword present

`writeto` (*name*, *output_verify='exception'*, *overwrite=False*, *checksum=False*)

Write the HDU to a new file. This is a convenience method to provide a user easier output interface if only one HDU needs to be written to a file.

Parameters: **name** : *file path, file object or file-like object*

Output FITS file. If the file object is already opened, it must be opened in a writeable mode.

output_verify : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `+warn`, or `+exception" (e.g. ``"fix+warn"`). See Verification Options for more info.

overwrite : *bool, optional*

If `True`, overwrite the output file if it exists. Raises an `OSError` if `False` and the output file exists. Default is `False`.

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

checksum : *bool*

When **`True`** adds both `DATASUM` and `CHECKSUM` cards to the header of the HDU when written to the file.

GroupsHDU

class `astropy.io.fits.` **`GroupsHDU`**(*data=None*, *header=None*)

Bases: **`astropy.io.fits.PrimaryHDU`**, **`astropy.io.fits.hdu.table._TableLikeHDU`**

FITS Random Groups HDU class.

See the Random Access Groups section in the Astropy documentation for more details on working with this type of HDU.

Construct a primary HDU.

Parameters: **data** : *array or DELAYED, optional*

The data in the HDU.

header : *`Header`, optional*

The header to be used (as a template). If `header` is `None`, a minimal header will be provided.

do_not_scale_image_data : *bool, optional*

If `True`, image data is not scaled using BSCALE/BZERO values when read. (default: False)

ignore_blank : *bool, optional*

If `True`, the BLANK header keyword will be ignored if present. Otherwise, pixels equal to this value will be replaced with NaNs. (default: False)

uint : *bool, optional*

Interpret signed integer data where `BZERO` is the central value and `BSCALE == 1` as unsigned integer data. For example, `int16` data with `BZERO = 32768` and `BSCALE = 1` would be treated as `uint16` data. (default: True)

scale_back : *bool, optional*

If `True`, when saving changes to a file that contained scaled image data, restore the data to the original type and reapply the original BSCALE/BZERO values. This could lead to loss of accuracy if scaling back to integer values after performing floating point operations on the data. Pseudo-unsigned integers are automatically rescaled unless scale_back is explicitly set to `False`. (default: None)

`add_checksum`(*when=None*, *override_datasum=False*, *checksum_keyword='CHECKSUM'*, *datasum_keyword='DATASUM'*)

Add the `CHECKSUM` and `DATASUM` cards to this HDU with the values set to the checksum calculated for the HDU and the data respectively. The addition of the `DATASUM` card may be overridden.

Parameters: **when** : *str, optional*

comment string for the cards; by default the comments will represent the time when the checksum was calculated

override_datasum : *bool, optional*

add the `CHECKSUM` card only

checksum_keyword : *str, optional*

The name of the header keyword to store the checksum value in; this is typically 'CHECKSUM' per convention, but there exist use cases in which a different keyword should be used

datasum_keyword : *str, optional*

See `checksum_keyword`

Notes

For testing purposes, first call **`add_datasum`** with a `when` argument, then call **`add_checksum`** with a `when` argument and `override_datasum` set to **`True`**. This will provide consistent comments for both cards and enable the generation of a `CHECKSUM` card with a consistent value.

`add_datasum`(*when=None*, *datasum_keyword='DATASUM'*)

Add the `DATASUM` card to this HDU with the value set to the checksum calculated for the data.

Parameters: **when** : *str, optional*

Comment string for the card that by default represents the time when the checksum was calculated

datasum_keyword : *str, optional*

The name of the header keyword to store the datasum value in; this is typically 'DATASUM' per convention, but there exist use cases in which a different keyword should be used

Returns: **checksum** : *int*

The calculated datasum

Notes

For testing purposes, provide a `when` argument to enable the comment value in the card to remain consistent. This will enable the generation of a `CHECKSUM` card with a consistent value.

property **`columns`**

The `ColDefs` objects describing the columns in this table.

`copy` ()

Make a copy of the HDU, both header and data are copied.

property `data`

The data of a random group FITS file will be like a binary table's data.

`filebytes` ()

Calculates and returns the number of bytes that this HDU will write to a file.

`fileinfo` ()

Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the `HDUList`.

Returns: dict or None

The dictionary details information about the locations of this HDU within an associated file. Returns `None` when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
filemode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
hdrLoc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

classmethod `from_columns` (*columns*, *header=None*, *nrows=0*, *fill=False*, *character_as_bytes=False*, ***kwargs*)

Given either a `ColDefs` object, a sequence of `Column` objects, or another table HDU or table data (a `FITS_rec` or multi-field `numpy.ndarray` or `numpy.recarray` object, return a new table HDU of the class this method was called on using the column definition from the input.

See also `FITS_rec.from_columns`.

Parameters: **columns** : *sequence of `Column`, `ColDefs`, or other*

The columns from which to create the table data, or an object with a column-like structure from which a `ColDefs` can be instantiated. This includes an existing `BinTableHDU` or `TableHDU`, or a `numpy.recarray` to give some examples.

If these columns have data arrays attached that data may be used in initializing the new table. Otherwise the input columns will be used as a template for a new table with the requested number of rows.

header : *`Header`*

An optional `Header` object to instantiate the new HDU yet. Header keywords specifically related to defining the table structure (such as the "TXXXn" keywords like TTYPEn) will be overridden by the supplied column definitions, but all other informational and data model-specific keywords are kept.

nrows : *int*

Number of rows in the new table. If the input columns have data associated with them, the size of the largest input column is used. Otherwise the default is 0.

fill : *bool*

If `True`, will fill all cells with zeros or blanks. If `False`, copy the data from input, undefined cells will still be filled with zeros/blanks.

character_as_bytes : *bool*

Whether to return bytes for string columns when accessed from the HDU. By default this is `False` and (unicode) strings are returned, but for large tables this may use up a lot of memory.

Notes

Any additional keyword arguments accepted by the HDU class's `__init__` may also be passed in as keyword arguments.

classmethod **`fromstring`**(*data*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Creates a new HDU object of the appropriate type from a string containing the HDU's entire header and, optionally, its data.

Note: When creating a new HDU from a string without a backing file object, the data of that HDU may be read-only. It depends on whether the underlying string was an immutable Python str/bytes object, or some kind of read-write

memory buffer such as a `memoryview`.

Parameters: **data** : *str, bytearray, memoryview, ndarray*

A byte string containing the HDU's header and data.

checksum : *bool, optional*

Check the HDU's checksum and/or datasum.

ignore_missing_end : *bool, optional*

Ignore a missing end card in the header data. Note that without the end card the end of the header may be ambiguous and resulted in a corrupt HDU. In this case the assumption is that the first 2880 block that does not begin with valid FITS header data is the beginning of the data.

kwargs : *optional*

May consist of additional keyword arguments specific to an HDU type–these correspond to keywords recognized by the constructors of different HDU classes such as `PrimaryHDU`, `ImageHDU`, or `BinTableHDU`. Any unrecognized keyword arguments are simply ignored.

classmethod **`match_header`**(*header*)

_ImageBaseHDU is sort of an abstract class for HDUs containing image data (as opposed to table data) and should never be used directly.

property **`parnames`**

The names of the group parameters as described by the header.

classmethod **`readfrom`**(*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with `open()` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto()`.

Parameters: **fileobj** : *file object or file-like object*

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : *bool*

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : *bool*

Do not issue an exception when opening a file that is missing an `END` card in the last header.

`req_cards`(*keyword*, *pos*, *test*, *fix_value*, *option*, *errlist*)

Check the existence, location, and value of a required `Card`.

Parameters: **keyword** : *str*

The keyword to validate

pos : *int, callable*

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument–the actual position of the keyword–and return `True` or `False`. This can be used for custom evaluation. For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : *callable*

This should be a callable (generally a function) that is passed the value of the given keyword and returns `True` or `False`. This can be used to validate the value associated with the given keyword.

fix_value : *str, int, float, complex, bool, None*

A valid value for a FITS keyword to to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If `None`, there is no replacement value and the keyword is unfixable.

option : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `+warn`, or `+exception" (e.g. ``"fix+warn"`). See Verification Options for more info.

errlist : *list*

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to `req_cards`.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

`run_option` (*option='warn'*, *err_text=''*, *fix_text='Fixed.'*, *fix=None*, *fixable=True*)

Execute the verification with selected option.

`scale` (*type=None*, *option='old'*, *bscale=None*, *bzero=None*)

Scale image data by using `BSCALE` / `BZERO`.

Call to this method will scale **`data`** and update the keywords of `BSCALE` and `BZERO` in the HDU's header. This method should only be used right before writing to the output file, as the data will be scaled and is therefore not very usable after the call.

Parameters:

type : *str, optional*

destination data type, use a string representing a numpy dtype name, (e.g. `'uint8'`, `'int16'`, `'float32'` etc.). If is **`None`**, use the current data type.

option : *str, optional*

How to scale the data: `"old"` uses the original `BSCALE` and `BZERO` values from when the data was read/created (defaulting to 1 and 0 if they don't exist). For integer data only, `"minmax"` uses the minimum and maximum of the data to scale. User-specified `bscale` / `bzero` values always take precedence.

bscale, bzero : *int, optional*

User-specified `BSCALE` and `BZERO` values

property **`section`**

Access a section of the image array without loading the entire array into memory. The **`Section`** object returned by this attribute is not meant to be used directly by itself. Rather, slices of the section return the appropriate slice of the data, and loads *only* that section into memory.

Sections are mostly obsoleted by memmap support, but should still be used to deal with very large scaled images. See the Data Sections section of the Astropy documentation for more details.

property **`shape`**

Shape of the image array–should be equivalent to `self.data.shape`.

property **`size`**

Returns the size (in bytes) of the HDU's data part.

`update_header` ()

Update the header keywords to agree with the data.

`verify` (*option='warn'*)

Verify all values in the instance.

Parameters: **option** : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `"+warn"`, or `"+exception"` (e.g. `"fix+warn"`). See Verification Options for more info.

`verify_checksum` ()

Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU CHECKSUM.

Returns: **valid** : *int*

- 0 - failure
- 1 - success
- 2 - no `CHECKSUM` keyword present

`verify_datasum` ()

Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

Returns: **valid** : *int*

- 0 - failure
- 1 - success
- 2 - no `DATASUM` keyword present

`writeto` (*name*, *output_verify='exception'*, *overwrite=False*, *checksum=False*)

Write the HDU to a new file. This is a convenience method to provide a user easier output interface if only one HDU needs to be written to a file.

Parameters: **name** : *file path, file object or file-like object*

Output FITS file. If the file object is already opened, it must be opened in a writeable mode.

output_verify : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `+warn`, or `+exception" (e.g. ``"fix+warn"`). See Verification Options for more info.

overwrite : *bool, optional*

If `True`, overwrite the output file if it exists. Raises an `OSError` if `False` and the output file exists. Default is `False`.

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

checksum : *bool*

When **`True`** adds both `DATASUM` and `CHECKSUM` cards to the header of the HDU when written to the file.

GroupData

class `astropy.io.fits.` **`GroupData`**(*input=None*, *bitpix=None*, *pardata=None*, *parnames=[]*, *bscale=None*, *bzero=None*, *parbscales=None*, *parbzeros=None*)

Bases: **`astropy.io.fits.FITS_rec`**

Random groups data object.

Allows structured access to FITS Group data in a manner analogous to tables.

Parameters: **input** : *array or FITS_rec instance*

input data, either the group data itself (a **numpy.ndarray**) or a record array (**FITS_rec**) which will contain both group parameter info and the data. The rest of the arguments are used only for the first case.

bitpix : *int*

data type as expressed in FITS `BITPIX` value (8, 16, 32, 64, -32, or -64)

pardata : *sequence of arrays*

parameter data, as a list of (numeric) arrays.

parnames : *sequence of str*

list of parameter names.

bscale : *int*

`BSCALE` of the data

bzero : *int*

`BZERO` of the data

parbscales : *sequence of int*

list of bscales for the parameters

parbzeros : *sequence of int*

list of bzeros for the parameters

property **`data`**

The raw group data represented as a multi-dimensional **numpy.ndarray** array.

`par`(*parname*)

Get the group parameter values.

Group

class `astropy.io.fits.`**`Group`**(*input*, *row=0*, *start=None*, *end=None*, *step=None*, *base=None*)

Bases: **astropy.io.fits.FITS_record**

One group of the random group data.

Parameters: **input** : *array*

The array to wrap.

row : *int, optional*

The starting logical row of the array.

start : *int, optional*

The starting column in the row associated with this object. Used for subsetting the columns of the `FITS_rec` object.

end : *int, optional*

The ending column in the row associated with this object. Used for subsetting the columns of the `FITS_rec` object.

`par` (*parname*)

Get the group parameter value.

`setpar` (*parname*, *value*)

Set the group parameter value.

StreamingHDU

class `astropy.io.fits.`**`StreamingHDU`**(*name*, *header*)

Bases: `object`

A class that provides the capability to stream data to a FITS file instead of requiring data to all be written at once.

The following pseudocode illustrates its use:

```
header = astropy.io.fits.Header()

for all the cards you need in the header:
    header[key] = (value, comment)

shdu = astropy.io.fits.StreamingHDU('filename.fits', header)

for each piece of data:
    shdu.write(data)

shdu.close()
```

Construct a `StreamingHDU` object given a file name and a header.

Parameters: **name** : *file path, file object, or file like object*

The file to which the header and data will be streamed. If opened, the file object must be opened in a writeable binary mode such as 'wb' or 'ab+'.

header : ***Header*** *instance*

The header object associated with the data to be written to the file.

Notes

The file will be opened and the header appended to the end of the file. If the file does not already exist, it will be created, and if the header represents a Primary header, it will be written to the beginning of the file. If the file does not exist and the provided header is not a Primary header, a default Primary HDU will be inserted at the beginning of the file and the provided header will be added as the first extension. If the file does already exist, but the provided header represents a Primary header, the header will be modified to an image extension header and appended to the end of the file.

`close` ()

Close the physical FITS file.

property **`size`**

Return the size (in bytes) of the data portion of the HDU.

`write` (*data*)

Write the given data to the stream.

Parameters: **data** : *ndarray*

Data to stream to the file.

Returns: **writecomplete** : *int*

Flag that when `True` indicates that all of the required data has been written to the stream.

Notes

Only the amount of data specified in the header provided to the class constructor may be written to the stream. If the provided data would cause the stream to overflow, an `OSError` exception is raised and the data is not written. Once sufficient data has been written to the stream to satisfy the amount specified in the header, the stream is padded to fill a complete FITS block and no more data will be accepted. An attempt to write more data after the stream has been filled will raise an `OSError` exception. If the dtype of the input data does not match what is expected by the header, a `TypeError` exception is raised.

Headers

Header

class `astropy.io.fits.` **`Header`** (*cards=[]*, *copy=False*)

Bases: `object`

FITS header class. This class exposes both a dict-like interface and a list-like interface to FITS headers.

The header may be indexed by keyword and, like a dict, the associated value will be returned. When the header contains cards with duplicate keywords, only the value of the first card with the given keyword will be returned. It is also possible to use a 2-tuple as the index in the form (keyword, n)–this returns the n-th value with that keyword, in the case where there are duplicate keywords.

For example:

```
>>> header['NAXIS']
0
>>> header[('FOO', 1)]  # Return the value of the second FOO keyword
'foo'
```

The header may also be indexed by card number:

```
>>> header[0]  # Return the value of the first card in the header
'T'
```

Commentary keywords such as HISTORY and COMMENT are special cases: When indexing the Header object with either 'HISTORY' or 'COMMENT' a list of all the HISTORY/COMMENT values is returned:

```
>>> header['HISTORY']
This is the first history entry in this header.
This is the second history entry in this header.
...
```

See the Astropy documentation for more details on working with headers.

Notes

Although FITS keywords must be exclusively upper case, retrieving an item in a **`Header`** object is case insensitive.

Construct a **`Header`** from an iterable and/or text file.

Parameters: **cards** : *A list of* ***`Card`*** *objects, optional*

The cards to initialize the header with. Also allowed are other **`Header`** (or **`dict`**-like) objects.

Changed in version 1.2: Allowed `cards` to be a **`dict`**-like object.

copy : *bool, optional*

If `True` copies the `cards` if they were another **`Header`** instance. Default is `False`.

New in version 1.3.

`add_blank`(*value=''*, *before=None*, *after=None*)

Add a blank card.

Parameters: **value** : *str, optional*

Text to be added.

before : *str or int, optional*

Same as in **`Header.update`**

after : *str or int, optional*

Same as in **`Header.update`**

`add_comment`(*value*, *before=None*, *after=None*)

Add a `COMMENT` card.

Parameters: **value** : *str*

Text to be added.

before : *str or int, optional*

Same as in **`Header.update`**

after : *str or int, optional*

Same as in **`Header.update`**

`add_history`(*value*, *before=None*, *after=None*)

Add a `HISTORY` card.

Parameters: **value** : *str*

History text to be added.

before : *str or int, optional*

Same as in `Header.update`

after : *str or int, optional*

Same as in `Header.update`

`append` (*card=None*, *useblanks=True*, *bottom=False*, *end=False*)

Appends a new keyword+value card to the end of the Header, similar to `list.append`.

By default if the last cards in the Header have commentary keywords, this will append the new keyword before the commentary (unless the new keyword is also commentary).

Also differs from `list.append` in that it can be called with no arguments: In this case a blank card is appended to the end of the Header. In the case all the keyword arguments are ignored.

Parameters: **card** : *str, tuple*

A keyword or a (keyword, value, [comment]) tuple representing a single header card; the comment is optional in which case a 2-tuple may be used

useblanks : *bool, optional*

If there are blank cards at the end of the Header, replace the first blank card so that the total number of cards in the Header does not increase. Otherwise preserve the number of blank cards.

bottom : *bool, optional*

If True, instead of appending after the last non-commentary card, append after the last non-blank card.

end : *bool, optional*

If True, ignore the useblanks and bottom options, and append at the very end of the Header.

property `cards`

The underlying physical cards that make up this Header; it can be looked at, but it should not be modified directly.

`clear` ()

Remove all cards from the header.

property **`comments`**

View the comments associated with each keyword, if any.

For example, to see the comment on the NAXIS keyword:

```
>>> header.comments['NAXIS']
number of data axes
```

Comments can also be updated through this interface:

```
>>> header.comments['NAXIS'] = 'Number of data axes'
```

`copy` (*strip=False*)

Make a copy of the `Header`.

Changed in version 1.3: `copy.copy` and `copy.deepcopy` on a `Header` will call this method.

Parameters: **strip** : *bool, optional*

If `True`, strip any headers that are specific to one of the standard HDU types, so that this header can be used in a different HDU.

Returns: header

A new `Header` instance.

`count` (*keyword*)

Returns the count of the given keyword in the header, similar to **`list.count`** if the Header object is treated as a list of keywords.

Parameters: **keyword** : *str*

The keyword to count instances of in the header

`extend` (*cards*, *strip=True*, *unique=False*, *update=False*, *update_first=False*, *useblanks=True*, *bottom=False*, *end=False*)

Appends multiple keyword+value cards to the end of the header, similar to **`list.extend`**.

Parameters: **cards** : *iterable*

An iterable of (keyword, value, [comment]) tuples; see `Header.append`.

strip : *bool, optional*

Remove any keywords that have meaning only to specific types of HDUs, so that only more general keywords are added from extension Header or Card list (default: `True`).

unique : *bool, optional*

If `True`, ensures that no duplicate keywords are appended; keywords already in this header are simply discarded. The exception is commentary keywords (COMMENT, HISTORY, etc.): they are only treated as duplicates if their values match.

update : *bool, optional*

If `True`, update the current header with the values and comments from duplicate keywords in the input header. This supersedes the `unique` argument. Commentary keywords are treated the same as if `unique=True`.

update_first : *bool, optional*

If the first keyword in the header is 'SIMPLE', and the first keyword in the input header is 'XTENSION', the 'SIMPLE' keyword is replaced by the 'XTENSION' keyword. Likewise if the first keyword in the header is 'XTENSION' and the first keyword in the input header is 'SIMPLE', the 'XTENSION' keyword is replaced by the 'SIMPLE' keyword. This behavior is otherwise dumb as to whether or not the resulting header is a valid primary or extension header. This is mostly provided to support backwards compatibility with the old `Header.fromTxtFile` method, and only applies if `update=True`.

useblanks, bottom, end : *bool, optional*

These arguments are passed to `Header.append()` while appending new cards to the header.

classmethod **`fromfile`**(*fileobj*, *sep=''*, *endcard=True*, *padding=True*)

Similar to `Header.fromstring()`, but reads the header string from a given file-like object or filename.

Parameters: **fileobj** : *str, file-like*

A filename or an open file-like object from which a FITS header is to be read. For open file handles the file pointer must be at the beginning of the header.

sep : *str, optional*

The string separating cards from each other, such as a newline. By default there is no card separator (as is the case in a raw FITS file).

endcard : *bool, optional*

If True (the default) the header must end with an END card in order to be considered valid. If an END card is not found an `OSError` is raised.

padding : *bool, optional*

If True (the default) the header will be required to be padded out to a multiple of 2880, the FITS header block size. Otherwise any padding, or lack thereof, is ignored.

Returns: header

A new `Header` instance.

classmethod `fromkeys` (*iterable*, *value=None*)

Similar to `dict.fromkeys()`–creates a new `Header` from an iterable of keywords and an optional default value.

This method is not likely to be particularly useful for creating real world FITS headers, but it is useful for testing.

Parameters: **iterable**

Any iterable that returns strings representing FITS keywords.

value : *optional*

A default value to assign to each keyword; must be a valid type for FITS keywords.

Returns: header

A new `Header` instance.

classmethod `fromstring` (*data*, *sep=''*)

Creates an HDU header from a byte string containing the entire header data.

Parameters: **data** : *str or bytes*

String or bytes containing the entire header. In the case of bytes they will be decoded using latin-1 (only plain ASCII characters are allowed in FITS headers but latin-1 allows us to retain any invalid bytes that might appear in malformatted FITS files).

sep : *str, optional*

The string separating cards from each other, such as a newline. By default there is no card separator (as is the case in a raw FITS file). In general this is only used in cases where a header was printed as text (e.g. with newlines after each card) and you want to create a new `Header` from it by copy/pasting.

Returns: header

A new `Header` instance.

Examples

```
>>> from astropy.io.fits import Header
>>> hdr = Header({'SIMPLE': True})
>>> Header.fromstring(hdr.tostring()) == hdr
True
```

If you want to create a `Header` from printed text it's not necessary to have the exact binary structure as it would appear in a FITS file, with the full 80 byte card length. Rather, each "card" can end in a newline and does not have to be padded out to a full card length as long as it "looks like" a FITS header:

```
>>> hdr = Header.fromstring("""\
... SIMPLE  =                    T / conforms to FITS standard
... BITPIX  =                    8 / array data type
... NAXIS   =                    0 / number of array dimensions
... EXTEND  =                    T
... """, sep='\n')
>>> hdr['SIMPLE']
True
>>> hdr['BITPIX']
8
>>> len(hdr)
4
```

classmethod **`fromtextfile`**(*fileobj*, *endcard=False*)

Read a header from a simple text file or file-like object.

Equivalent to:

```
>>> Header.fromfile(fileobj, sep='\n', endcard=False,
...                 padding=False)
```

See also

`fromfile`

`get` (*key*, *default=None*)

Similar to `dict.get()`–returns the value associated with keyword in the header, or a default value if the keyword is not found.

Parameters: **key** : *str*

A keyword that may or may not be in the header.

default : *optional*

A default value to return if the keyword is not found in the header.

Returns: value

The value associated with the given keyword, or the default value if the keyword is not in the header.

`index` (*keyword*, *start=None*, *stop=None*)

Returns the index if the first instance of the given keyword in the header, similar to `list.index` if the Header object is treated as a list of keywords.

Parameters: **keyword** : *str*

The keyword to look up in the list of all keywords in the header

start : *int, optional*

The lower bound for the index

stop : *int, optional*

The upper bound for the index

`insert` (*key*, *card*, *useblanks=True*, *after=False*)

Inserts a new keyword+value card into the Header at a given location, similar to `list.insert`.

Parameters: **key** : *int, str, or tuple*

The index into the list of header keywords before which the new keyword should be inserted, or the name of a keyword before which the new keyword should be inserted. Can also accept a (keyword, index) tuple for inserting around duplicate keywords.

card : *str, tuple*

A keyword or a (keyword, value, [comment]) tuple; see `Header.append`

useblanks : *bool, optional*

If there are blank cards at the end of the Header, replace the first blank card so that the total number of cards in the Header does not increase. Otherwise preserve the number of blank cards.

after : *bool, optional*

If set to `True`, insert *after* the specified index or keyword, rather than before it. Defaults to `False`.

`items` ()

Like `dict.items()`.

`keys` ()

Like `dict.keys()`–iterating directly over the `Header` instance has the same behavior.

`pop` (**args*)

Works like `list.pop()` if no arguments or an index argument are supplied; otherwise works like `dict.pop()`.

`popitem` ()

Similar to `dict.popitem()`.

`remove` (*keyword*, *ignore_missing=False*, *remove_all=False*)

Removes the first instance of the given keyword from the header similar to `list.remove` if the Header object is treated as a list of keywords.

Parameters: **keyword** : *str*

The keyword of which to remove the first instance in the header.

ignore_missing : *bool, optional*

When True, ignores missing keywords. Otherwise, if the keyword is not present in the header a KeyError is raised.

remove_all : *bool, optional*

When True, all instances of keyword will be removed. Otherwise only the first instance of the given keyword is removed.

`rename_keyword` (*oldkeyword*, *newkeyword*, *force=False*)

Rename a card's keyword in the header.

Parameters: **oldkeyword** : *str or int*

Old keyword or card index

newkeyword : *str*

New keyword

force : *bool, optional*

When `True`, if the new keyword already exists in the header, force the creation of a duplicate keyword. Otherwise a `ValueError` is raised.

`set` (*keyword*, *value=None*, *comment=None*, *before=None*, *after=None*)

Set the value and/or comment and/or position of a specified keyword.

If the keyword does not already exist in the header, a new keyword is created in the specified position, or appended to the end of the header if no position is specified.

This method is similar to `Header.update()` prior to Astropy v0.1.

Note

It should be noted that `header.set(keyword, value)` and `header.set(keyword, value, comment)` are equivalent to `header[keyword] = value` and `header[keyword] = (value, comment)` respectively.

New keywords can also be inserted relative to existing keywords using, for example:

```
>>> header.insert('NAXIS1', ('NAXIS', 2, 'Number of axes'))
```

to insert before an existing keyword, or:

```
>>> header.insert('NAXIS', ('NAXIS1', 4096), after=True)
```

to insert after an existing keyword.

The only advantage of using **Header.set()** is that it easily replaces the old usage of **Header.update()** both conceptually and in terms of function signature.

Parameters: **keyword** : *str*

A header keyword

value : *str, optional*

The value to set for the given keyword; if None the existing value is kept, but '' may be used to set a blank value

comment : *str, optional*

The comment to set for the given keyword; if None the existing comment is kept, but `''` may be used to set a blank comment

before : *str, int, optional*

Name of the keyword, or index of the **Card** before which this card should be located in the header. The argument `before` takes precedence over `after` if both specified.

after : *str, int, optional*

Name of the keyword, or index of the **Card** after which this card should be located in the header.

setdefault(*key*, *default=None*)

Similar to **dict.setdefault()**.

tofile(*fileobj*, *sep=''*, *endcard=True*, *padding=True*, *overwrite=False*)

Writes the header to file or file-like object.

By default this writes the header exactly as it would be written to a FITS file, with the END card included and padding to the next multiple of 2880 bytes. However, aspects of this may be controlled.

Parameters: **fileobj** : *str, file, optional*

Either the pathname of a file, or an open file handle or file-like object

sep : *str, optional*

The character or string with which to separate cards. By default there is no separator, but one could use `'\\n'`, for example, to separate each card with a new line

endcard : *bool, optional*

If **`True`** (default) adds the END card to the end of the header string

padding : *bool, optional*

If **`True`** (default) pads the string with spaces out to the next multiple of 2880 characters

overwrite : *bool, optional*

If `True`, overwrite the output file if it exists. Raises an `OSError` if `False` and the output file exists. Default is `False`.

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

`tostring` (*sep='', endcard=True, padding=True*)

Returns a string representation of the header.

By default this uses no separator between cards, adds the END card, and pads the string with spaces to the next multiple of 2880 bytes. That is, it returns the header exactly as it would appear in a FITS file.

Parameters: **sep** : *str, optional*

The character or string with which to separate cards. By default there is no separator, but one could use `'\\n'`, for example, to separate each card with a new line

endcard : *bool, optional*

If True (default) adds the END card to the end of the header string

padding : *bool, optional*

If True (default) pads the string with spaces out to the next multiple of 2880 characters

Returns: **s** : *str*

A string representing a FITS header.

totextfile (*fileobj*, *endcard=False*, *overwrite=False*)

Write the header as text to a file or a file-like object.

Equivalent to:

```
>>> Header.tofile(fileobj, sep='\n', endcard=False,
...               padding=False, overwrite=overwrite)
```

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

See also

`tofile`

update (**args*, ***kwargs*)

Update the Header with new keyword values, updating the values of existing keywords and appending new keywords otherwise; similar to `dict.update`.

`update` accepts either a dict-like object or an iterable. In the former case the keys must be header keywords and the values may be either scalar values or (value, comment) tuples. In the case of an iterable the items must be (keyword, value) tuples or (keyword, value, comment) tuples.

Arbitrary arguments are also accepted, in which case the update() is called again with the kwargs dict as its only argument. That is,

```
>>> header.update(NAXIS1=100, NAXIS2=100)
```

is equivalent to:

```
header.update({'NAXIS1': 100, 'NAXIS2': 100})
```

Warning

As this method works similarly to `dict.update` it is very different from the `Header.update()` method in Astropy v0.1. Use of the old API was **deprecated** for a long time and is now removed. Most uses of the old API can be replaced as follows:

- Replace

  ```
  header.update(keyword, value)
  ```

  with

```
header[keyword] = value
```

- Replace

  ```
  header.update(keyword, value, comment=comment)
  ```

  with

  ```
  header[keyword] = (value, comment)
  ```

- Replace

  ```
  header.update(keyword, value, before=before_keyword)
  ```

  with

  ```
  header.insert(before_keyword, (keyword, value))
  ```

- Replace

  ```
  header.update(keyword, value, after=after_keyword)
  ```

  with

  ```
  header.insert(after_keyword, (keyword, value),
                after=True)
  ```

See also **`Header.set()`** which is a new method that provides an interface similar to the old `Header.update()` and may help make transition a little easier.

`values` ()

Like **`dict.values()`**.

Cards

Card

class `astropy.io.fits.` **`Card`** (*keyword=None*, *value=None*, *comment=None*, ***kwargs*)

Bases: **astropy.io.fits.verify._Verify**

property **comment**

Get the comment attribute from the card image if not already set.

property **field_specifier**

The field-specifier of record-valued keyword cards; always **None** on normal cards.

classmethod **fromstring**(*image*)

Construct a **Card** object from a (raw) string. It will pad the string if it is not the length of a card image (80 columns). If the card image is longer than 80 columns, assume it contains `CONTINUE` card(s).

property **image**

The card "image", that is, the 80 byte character string that represents this card in an actual FITS header.

property **is_blank**

True if the card is completely blank–that is, it has no keyword, value, or comment. It appears in the header as 80 spaces.

Returns **False** otherwise.

property **keyword**

Returns the keyword name parsed from the card image.

length *= 80*

The length of a Card image; should always be 80 for valid FITS files.

classmethod **normalize_keyword**(*keyword*)

classmethod to convert a keyword value that may contain a field-specifier to uppercase. The effect is to raise the key to uppercase and leave the field specifier in its original case.

Parameters: **keyword** : *or str*

A keyword value or a `keyword.field-specifier` value

property **rawkeyword**

On record-valued keyword cards this is the name of the standard <= 8 character FITS keyword that this RVKC is stored in. Otherwise it is the card's normal keyword.

property **rawvalue**

On record-valued keyword cards this is the raw string value in the `<field-specifier>: <value>` format stored in the card in order to represent a RVKC. Otherwise it is the card's normal value.

`run_option` (*option='warn'*, *err_text=''*, *fix_text='Fixed.'*, *fix=None*, *fixable=True*)

Execute the verification with selected option.

property **`value`**

The value associated with the keyword stored in this card.

`verify` (*option='warn'*)

Verify all values in the instance.

Parameters: **option** : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `"+warn"`, or `"+exception"` (e.g. `"fix+warn"`). See Verification Options for more info.

Tables

BinTableHDU

class `astropy.io.fits.` **`BinTableHDU`** (*data=None*, *header=None*, *name=None*, *uint=False*, *ver=None*, *character_as_bytes=False*)

Bases: **`astropy.io.fits.hdu.table._TableBaseHDU`**

Binary table HDU class.

Parameters: **data** : *array,* `FITS_rec`*, or* `Table`

Data to be used.

header : `Header`

Header to be used.

name : *str*

Name to be populated in `EXTNAME` keyword.

uint : *bool, optional*

Set to `True` if the table contains unsigned integer columns.

ver : *int > 0 or None, optional*

The ver of the HDU, will be the value of the keyword `EXTVER`. If not given or None, it defaults to the value of the `EXTVER` card of the `header` or 1. (default: None)

character_as_bytes : *bool*

Whether to return bytes for string columns. By default this is `False` and (unicode) strings are returned, but this does not respect memory mapping and loads the whole column in memory when accessed.

`add_checksum`(*when=None*, *override_datasum=False*, *checksum_keyword='CHECKSUM'*, *datasum_keyword='DATASUM'*)

Add the `CHECKSUM` and `DATASUM` cards to this HDU with the values set to the checksum calculated for the HDU and the data respectively. The addition of the `DATASUM` card may be overridden.

Parameters: **when** : *str, optional*

comment string for the cards; by default the comments will represent the time when the checksum was calculated

override_datasum : *bool, optional*

add the `CHECKSUM` card only

checksum_keyword : *str, optional*

The name of the header keyword to store the checksum value in; this is typically 'CHECKSUM' per convention, but there exist use cases in which a different keyword should be used

datasum_keyword : *str, optional*

See `checksum_keyword`

Notes

For testing purposes, first call `add_datasum` with a `when` argument, then call `add_checksum` with a `when` argument and `override_datasum` set to `True`. This will provide consistent comments for both cards and enable the generation of a `CHECKSUM` card with a consistent value.

`add_datasum` (*when=None*, *datasum_keyword='DATASUM'*)

Add the `DATASUM` card to this HDU with the value set to the checksum calculated for the data.

Parameters:

when : *str, optional*

Comment string for the card that by default represents the time when the checksum was calculated

datasum_keyword : *str, optional*

The name of the header keyword to store the datasum value in; this is typically 'DATASUM' per convention, but there exist use cases in which a different keyword should be used

Returns:

checksum : *int*

The calculated datasum

Notes

For testing purposes, provide a `when` argument to enable the comment value in the card to remain consistent. This will enable the generation of a `CHECKSUM` card with a consistent value.

property **`columns`**

The `ColDefs` objects describing the columns in this table.

`copy` ()

Make a copy of the table HDU, both header and data are copied.

`dump` (*datafile=None*, *cdfile=None*, *hfile=None*, *overwrite=False*)

Dump the table HDU to a file in ASCII format. The table may be dumped in three separate files, one containing column definitions, one containing header parameters, and one for table data.

Parameters: **datafile** : *file path, file object or file-like object, optional*

Output data file. The default is the root name of the fits file associated with this HDU appended with the extension `.txt`.

cdfile : *file path, file object or file-like object, optional*

Output column definitions file. The default is `None`, no column definitions output is produced.

hfile : *file path, file object or file-like object, optional*

Output header parameters file. The default is `None`, no header parameters output is produced.

overwrite : *bool, optional*

If `True`, overwrite the output file if it exists. Raises an `OSError` if `False` and the output file exists. Default is `False`.

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

Notes

The primary use for the `dump` method is to allow viewing and editing the table data and parameters in a standard text editor. The `load` method can be used to create a new table from the three plain text (ASCII) files.

- **datafile:** Each line of the data file represents one row of table data. The data is output one column at a time in column order. If a column contains an array, each element of the column array in the current row is output before moving on to the next column. Each row ends with a new line.

  Integer data is output right-justified in a 21-character field followed by a blank. Floating point data is output right justified using 'g' format in a 21-character field with 15 digits of precision, followed by a blank. String data that does not contain whitespace is output left-justified in a field whose width matches the width specified in the `TFORM` header parameter for the column, followed by a blank. When the string data contains whitespace characters, the string is enclosed in quotation marks (`""`). For the last data element in a row, the trailing blank in the field is replaced by a new line character.

  For column data containing variable length arrays ('P' format), the array data is preceded by the string `'VLA_Length= '` and the integer length of the array for that row, left-justified in a 21-character field, followed by a blank.

  Note

This format does *not* support variable length arrays using the ('Q' format) due to difficult to overcome ambiguities. What this means is that this file format cannot support VLA columns in tables stored in files that are over 2 GB in size.

For column data representing a bit field ('X' format), each bit value in the field is output right-justified in a 21-character field as 1 (for true) or 0 (for false).

- **cdfile:** Each line of the column definitions file provides the definitions for one column in the table. The line is broken up into 8, sixteen-character fields. The first field provides the column name (`TTYPEn`). The second field provides the column format (`TFORMn`). The third field provides the display format (`TDISPn`). The fourth field provides the physical units (`TUNITn`). The fifth field provides the dimensions for a multidimensional array (`TDIMn`). The sixth field provides the value that signifies an undefined value (`TNULLn`). The seventh field provides the scale factor (`TSCALn`). The eighth field provides the offset value (`TZEROn`). A field value of `""` is used to represent the case where no value is provided.
- **hfile:** Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

`filebytes` ()

Calculates and returns the number of bytes that this HDU will write to a file.

`fileinfo` ()

Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the `HDUList`.

Returns: dict or None

The dictionary details information about the locations of this HDU within an associated file. Returns `None` when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
filemode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
hdrLoc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

classmethod `from_columns`(*columns*, *header=None*, *nrows=0*, *fill=False*, *character_as_bytes=False*, ***kwargs*)

Given either a `ColDefs` object, a sequence of `Column` objects, or another table HDU or table data (a `FITS_rec` or multi-field `numpy.ndarray` or `numpy.recarray` object, return a new table HDU of the class this method was called on using the column definition from the input.

See also `FITS_rec.from_columns`.

Parameters:

columns : *sequence of `Column`, `ColDefs`, or other*

The columns from which to create the table data, or an object with a column-like structure from which a `ColDefs` can be instantiated. This includes an existing `BinTableHDU` or `TableHDU`, or a `numpy.recarray` to give some examples.

If these columns have data arrays attached that data may be used in initializing the new table. Otherwise the input columns will be used as a template for a new table with the requested number of rows.

header : *`Header`*

An optional `Header` object to instantiate the new HDU yet. Header keywords specifically related to defining the table structure (such as the “TXXXn” keywords like TTYPEn) will be overridden by the supplied column definitions, but all other informational and data model-specific keywords are kept.

nrows : *int*

Number of rows in the new table. If the input columns have data associated with them, the size of the largest input column is used. Otherwise the default is 0.

fill : *bool*

If `True`, will fill all cells with zeros or blanks. If `False`, copy the data from input, undefined cells will still be filled with zeros/blanks.

character_as_bytes : *bool*

Whether to return bytes for string columns when accessed from the HDU. By default this is `False` and (unicode) strings are returned, but for large tables this may use up a lot of memory.

Notes

Any additional keyword arguments accepted by the HDU class’s `__init__` may also be passed in as keyword arguments.

classmethod **`fromstring`**(*data*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Creates a new HDU object of the appropriate type from a string containing the HDU's entire header and, optionally, its data.

Note: When creating a new HDU from a string without a backing file object, the data of that HDU may be read-only. It depends on whether the underlying string was an immutable Python str/bytes object, or some kind of read-write memory buffer such as a `memoryview`.

Parameters: **data** : *str, bytearray, memoryview, ndarray*

A byte string containing the HDU's header and data.

checksum : *bool, optional*

Check the HDU's checksum and/or datasum.

ignore_missing_end : *bool, optional*

Ignore a missing end card in the header data. Note that without the end card the end of the header may be ambiguous and resulted in a corrupt HDU. In this case the assumption is that the first 2880 block that does not begin with valid FITS header data is the beginning of the data.

kwargs : *optional*

May consist of additional keyword arguments specific to an HDU type–these correspond to keywords recognized by the constructors of different HDU classes such as `PrimaryHDU`, `ImageHDU`, or `BinTableHDU`. Any unrecognized keyword arguments are simply ignored.

classmethod **`load`**(*datafile*, *cdfile=None*, *hfile=None*, *replace=False*, *header=None*)

Create a table from the input ASCII files. The input is from up to three separate files, one containing column definitions, one containing header parameters, and one containing column data.

The column definition and header parameters files are not required. When absent the column definitions and/or header parameters are taken from the header object given in the header argument; otherwise sensible defaults are inferred (though this mode is not recommended).

Parameters:

datafile : *file path, file object or file-like object*

Input data file containing the table data in ASCII format.

cdfile : *file path, file object, file-like object, optional*

Input column definition file containing the names, formats, display formats, physical units, multidimensional array dimensions, undefined values, scale factors, and offsets associated with the columns in the table. If `None`, the column definitions are taken from the current values in this object.

hfile : *file path, file object, file-like object, optional*

Input parameter definition file containing the header parameter definitions to be associated with the table. If `None`, the header parameter definitions are taken from the current values in this objects header.

replace : *bool, optional*

When `True`, indicates that the entire header should be replaced with the contents of the ASCII file instead of just updating the current header.

header : *`Header`, optional*

When the cdfile and hfile are missing, use this Header object in the creation of the new table and HDU. Otherwise this Header supersedes the keywords from hfile, which is only used to update values not present in this Header, unless `replace=True` in which this Header's values are completely replaced with the values from hfile.

Notes

The primary use for the `load` method is to allow the input of ASCII data that was edited in a standard text editor of the table data and parameters. The `dump` method can be used to create the initial ASCII files.

- **datafile:** Each line of the data file represents one row of table data. The data is output one column at a time in column order. If a column contains an array, each element of the column array in the current row is output before moving on to the next column. Each row ends with a new line.

  Integer data is output right-justified in a 21-character field followed by a blank. Floating point data is output right justified using 'g' format in a 21-character field with 15 digits of precision, followed by a blank. String data that does not contain whitespace is output left-justified in a field whose width matches the width specified in the `TFORM` header parameter for the column, followed by a blank. When the string data contains whitespace characters, the string is enclosed in quotation marks (`""`). For the last data element in a row, the trailing blank in the field is replaced by a new line character.

For column data containing variable length arrays ('P' format), the array data is preceded by the string `'VLA_Length= '` and the integer length of the array for that row, left-justified in a 21-character field, followed by a blank.

Note

This format does *not* support variable length arrays using the ('Q' format) due to difficult to overcome ambiguities. What this means is that this file format cannot support VLA columns in tables stored in files that are over 2 GB in size.

For column data representing a bit field ('X' format), each bit value in the field is output right-justified in a 21-character field as 1 (for true) or 0 (for false).

- **cdfile:** Each line of the column definitions file provides the definitions for one column in the table. The line is broken up into 8, sixteen-character fields. The first field provides the column name (`TTYPEn`). The second field provides the column format (`TFORMn`). The third field provides the display format (`TDISPn`). The fourth field provides the physical units (`TUNITn`). The fifth field provides the dimensions for a multidimensional array (`TDIMn`). The sixth field provides the value that signifies an undefined value (`TNULLn`). The seventh field provides the scale factor (`TSCALn`). The eighth field provides the offset value (`TZEROn`). A field value of `""` is used to represent the case where no value is provided.
- **hfile:** Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

classmethod **`match_header`** (*header*)

This is an abstract type that implements the shared functionality of the ASCII and Binary Table HDU types, which should be used instead of this.

classmethod **`readfrom`** (*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with **`open()`** which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with **`writeto()`**.

Parameters: **fileobj** : *file object or file-like object*

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : *bool*

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : *bool*

Do not issue an exception when opening a file that is missing an `END` card in the last header.

`req_cards`(*keyword*, *pos*, *test*, *fix_value*, *option*, *errlist*)

Check the existence, location, and value of a required `Card`.

Parameters: **keyword** : *str*

The keyword to validate

pos : *int, callable*

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument–the actual position of the keyword–and return **`True`** or **`False`**. This can be used for custom evaluation. For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : *callable*

This should be a callable (generally a function) that is passed the value of the given keyword and returns **`True`** or **`False`**. This can be used to validate the value associated with the given keyword.

fix_value : *str, int, float, complex, bool, None*

A valid value for a FITS keyword to to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If **`None`**, there is no replacement value and the keyword is unfixable.

option : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `+warn`, or `+exception" (e.g. ``"fix+warn"`). See Verification Options for more info.

errlist : *list*

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to **`req_cards`**.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

run_option(*option='warn', err_text='', fix_text='Fixed.', fix=None, fixable=True*)

Execute the verification with selected option.

property **size**

Size (in bytes) of the data portion of the HDU.

update()

Update header keywords to reflect recent changes of columns.

verify(*option='warn'*)

Verify all values in the instance.

Parameters: option : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `"+warn"`, or `"+exception"` (e.g. `"fix+warn"`). See Verification Options for more info.

verify_checksum()

Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU CHECKSUM.

Returns: valid : *int*

- 0 - failure
- 1 - success
- 2 - no `CHECKSUM` keyword present

verify_datasum()

Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

Returns: valid : *int*

- 0 - failure
- 1 - success
- 2 - no `DATASUM` keyword present

writeto(*name, output_verify='exception', overwrite=False, checksum=False*)

Works similarly to the normal writeto(), but prepends a default **`PrimaryHDU`** are required by extension HDUs (which cannot stand on their own).

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

TableHDU

class `astropy.io.fits.` **`TableHDU`**(*data=None*, *header=None*, *name=None*, *ver=None*, *character_as_bytes=False*)

Bases: **`astropy.io.fits.hdu.table._TableBaseHDU`**

FITS ASCII table extension HDU class.

Parameters:

data : *array or* ***`FITS_rec`***

Data to be used.

header : ***`Header`***

Header to be used.

name : *str*

Name to be populated in `EXTNAME` keyword.

ver : *int > 0 or None, optional*

The ver of the HDU, will be the value of the keyword `EXTVER`. If not given or None, it defaults to the value of the `EXTVER` card of the `header` or 1. (default: None)

character_as_bytes : *bool*

Whether to return bytes for string columns. By default this is **`False`** and (unicode) strings are returned, but this does not respect memory mapping and loads the whole column in memory when accessed.

`add_checksum`(*when=None*, *override_datasum=False*, *checksum_keyword='CHECKSUM'*, *datasum_keyword='DATASUM'*)

Add the `CHECKSUM` and `DATASUM` cards to this HDU with the values set to the checksum calculated for the HDU and the data respectively. The addition of the `DATASUM` card may be overridden.

Parameters: **when** : *str, optional*

comment string for the cards; by default the comments will represent the time when the checksum was calculated

override_datasum : *bool, optional*

add the `CHECKSUM` card only

checksum_keyword : *str, optional*

The name of the header keyword to store the checksum value in; this is typically 'CHECKSUM' per convention, but there exist use cases in which a different keyword should be used

datasum_keyword : *str, optional*

See `checksum_keyword`

Notes

For testing purposes, first call **`add_datasum`** with a `when` argument, then call **`add_checksum`** with a `when` argument and `override_datasum` set to **`True`**. This will provide consistent comments for both cards and enable the generation of a `CHECKSUM` card with a consistent value.

`add_datasum`(*when=None, datasum_keyword='DATASUM'*)

Add the `DATASUM` card to this HDU with the value set to the checksum calculated for the data.

Parameters: **when** : *str, optional*

Comment string for the card that by default represents the time when the checksum was calculated

datasum_keyword : *str, optional*

The name of the header keyword to store the datasum value in; this is typically 'DATASUM' per convention, but there exist use cases in which a different keyword should be used

Returns: **checksum** : *int*

The calculated datasum

Notes

For testing purposes, provide a `when` argument to enable the comment value in the card to remain consistent. This will enable the generation of a `CHECKSUM` card with a consistent value.

property **`columns`**

The `ColDefs` objects describing the columns in this table.

`copy` ()

Make a copy of the table HDU, both header and data are copied.

`filebytes` ()

Calculates and returns the number of bytes that this HDU will write to a file.

`fileinfo` ()

Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the `HDUList`.

Returns: dict or None

The dictionary details information about the locations of this HDU within an associated file. Returns `None` when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
filemode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
hdrLoc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

classmethod `from_columns` (*columns*, *header=None*, *nrows=0*, *fill=False*, *character_as_bytes=False*, ***kwargs*)

Given either a `ColDefs` object, a sequence of `Column` objects, or another table HDU or table data (a `FITS_rec` or multi-field `numpy.ndarray` or `numpy.recarray` object, return a new table HDU of the class this method was called on using the column definition from the input.

See also `FITS_rec.from_columns`.

Parameters: **columns** : *sequence of* `Column`, `ColDefs`, *or other*

The columns from which to create the table data, or an object with a column-like structure from which a `ColDefs` can be instantiated. This includes an existing `BinTableHDU` or `TableHDU`, or a `numpy.recarray` to give some examples.

If these columns have data arrays attached that data may be used in initializing the new table. Otherwise the input columns will be used as a template for a new table with the requested number of rows.

header : `Header`

An optional `Header` object to instantiate the new HDU yet. Header keywords specifically related to defining the table structure (such as the "TXXXn" keywords like TTYPEn) will be overridden by the supplied column definitions, but all other informational and data model-specific keywords are kept.

nrows : *int*

Number of rows in the new table. If the input columns have data associated with them, the size of the largest input column is used. Otherwise the default is 0.

fill : *bool*

If `True`, will fill all cells with zeros or blanks. If `False`, copy the data from input, undefined cells will still be filled with zeros/blanks.

character_as_bytes : *bool*

Whether to return bytes for string columns when accessed from the HDU. By default this is `False` and (unicode) strings are returned, but for large tables this may use up a lot of memory.

Notes

Any additional keyword arguments accepted by the HDU class's `__init__` may also be passed in as keyword arguments.

classmethod `fromstring`(*data*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Creates a new HDU object of the appropriate type from a string containing the HDU's entire header and, optionally, its data.

Note: When creating a new HDU from a string without a backing file object, the data of that HDU may be read-only. It depends on whether the underlying string was an immutable Python str/bytes object, or some kind of read-write

memory buffer such as a `memoryview`.

Parameters: **data** : *str, bytearray, memoryview, ndarray*

A byte string containing the HDU's header and data.

checksum : *bool, optional*

Check the HDU's checksum and/or datasum.

ignore_missing_end : *bool, optional*

Ignore a missing end card in the header data. Note that without the end card the end of the header may be ambiguous and resulted in a corrupt HDU. In this case the assumption is that the first 2880 block that does not begin with valid FITS header data is the beginning of the data.

kwargs : *optional*

May consist of additional keyword arguments specific to an HDU type–these correspond to keywords recognized by the constructors of different HDU classes such as `PrimaryHDU`, `ImageHDU`, or `BinTableHDU`. Any unrecognized keyword arguments are simply ignored.

classmethod **`match_header`**(*header*)

This is an abstract type that implements the shared functionality of the ASCII and Binary Table HDU types, which should be used instead of this.

classmethod **`readfrom`**(*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with `open()` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto()`.

Parameters: **fileobj** : *file object or file-like object*

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : *bool*

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : *bool*

Do not issue an exception when opening a file that is missing an `END` card in the last header.

`req_cards`(*keyword*, *pos*, *test*, *fix_value*, *option*, *errlist*)

Check the existence, location, and value of a required `Card`.

Parameters: **keyword** : *str*

The keyword to validate

pos : *int, callable*

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument–the actual position of the keyword–and return `True` or `False`. This can be used for custom evaluation. For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : *callable*

This should be a callable (generally a function) that is passed the value of the given keyword and returns `True` or `False`. This can be used to validate the value associated with the given keyword.

fix_value : *str, int, float, complex, bool, None*

A valid value for a FITS keyword to to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If `None`, there is no replacement value and the keyword is unfixable.

option : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `+warn`, or `+exception" (e.g. ``"fix+warn"`). See Verification Options for more info.

errlist : *list*

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to `req_cards`.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

`run_option` (*option='warn', err_text='', fix_text='Fixed.', fix=None, fixable=True*)

Execute the verification with selected option.

property **`size`**

Size (in bytes) of the data portion of the HDU.

`update` ()

Update header keywords to reflect recent changes of columns.

`verify` (*option='warn'*)

Verify all values in the instance.

Parameters: **option** : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `"+warn"`, or `"+exception"` (e.g. `"fix+warn"`). See Verification Options for more info.

`verify_checksum` ()

Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU CHECKSUM.

Returns: **valid** : *int*

- 0 - failure
- 1 - success
- 2 - no `CHECKSUM` keyword present

`verify_datasum` ()

Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

Returns: **valid** : *int*

- 0 - failure
- 1 - success
- 2 - no `DATASUM` keyword present

`writeto` (*name, output_verify='exception', overwrite=False, checksum=False*)

Works similarly to the normal writeto(), but prepends a default **`PrimaryHDU`** are required by extension HDUs (which cannot stand on their own).

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

`Column`

class `astropy.io.fits.` **`Column`**(*name=None*, *format=None*, *unit=None*, *null=None*, *bscale=None*, *bzero=None*, *disp=None*, *start=None*, *dim=None*, *array=None*, *ascii=None*, *coord_type=None*, *coord_unit=None*, *coord_ref_point=None*, *coord_ref_value=None*, *coord_inc=None*, *time_ref_pos=None*)

Bases: **`astropy.io.fits.util.NotifierMixin`**

Class which contains the definition of one column, e.g. `ttype`, `tform`, etc. and the array containing values for the column.

Construct a **`Column`** by specifying attributes. All attributes except `format` can be optional; see Column Creation and Creating an ASCII Table for more information regarding `TFORM` keyword.

Parameters: **name** : *str, optional*

column name, corresponding to `TTYPE` keyword

format : *str*

column format, corresponding to `TFORM` keyword

unit : *str, optional*

column unit, corresponding to `TUNIT` keyword

null : *str, optional*

null value, corresponding to `TNULL` keyword

bscale : *int-like, optional*

bscale value, corresponding to `TSCAL` keyword

bzero : *int-like, optional*

bzero value, corresponding to `TZERO` keyword

disp : *str, optional*

display format, corresponding to `TDISP` keyword

start : *int, optional*

column starting position (ASCII table only), corresponding to `TBCOL` keyword

dim : *str, optional*

column dimension corresponding to `TDIM` keyword

array : *iterable, optional*

a `list`, `numpy.ndarray` (or other iterable that can be used to initialize an ndarray) providing initial data for this column. The array will be automatically converted, if possible, to the data format of the column. In the case were non-trivial `bscale` and/or `bzero` arguments are given, the values in the array must be the *physical* values–that is, the values of column as if the scaling has already been applied (the array stored on the column object will then be converted back to its storage values).

ascii : *bool, optional*

set `True` if this describes a column for an ASCII table; this may be required to disambiguate the column format

coord_type : *str, optional*

coordinate/axis type corresponding to `TCTYP` keyword

coord_unit : *str, optional*

coordinate/axis unit corresponding to `TCUNI` keyword *property* `array`

The Numpy **`ndarray`** associated with this **`Column`**.

If the column was instantiated with an array passed to the `array` argument, this will return that array. However, if the column is later added to a table, such as via **`BinTableHDU.from_columns`** as is typically the case, this attribute will be updated to reference the associated field in the table, which may no longer be the same array.

property `ascii`

Whether this **`Column`** represents a column in an ASCII table.

`copy`()

Return a copy of this **`Column`**.

ColDefs

class `astropy.io.fits.` **`ColDefs`**(*input*, *ascii=False*)

Bases: **`astropy.io.fits.util.NotifierMixin`**

Column definitions class.

It has attributes corresponding to the **`Column`** attributes (e.g. **`ColDefs`** has the attribute `names` while **`Column`** has `name`). Each attribute in **`ColDefs`** is a list of corresponding attribute values from all **`Column`** objects.

Parameters: **input** : *sequence of* `Column`, `ColDefs`, *other*

An existing table HDU, an existing `ColDefs`, or any multi-field Numpy array or `numpy.recarray`.

ascii : *bool*

Use True to ensure that ASCII table columns are used.

`add_col`(*column*)

Append one `Column` to the column definition.

`change_attrib`(*col_name*, *attrib*, *new_value*)

Change an attribute (in the `KEYWORD_ATTRIBUTES` list) of a `Column`.

Parameters: **col_name** : *str or int*

The column name or index to change

attrib : *str*

The attribute name

new_value : *object*

The new value for the attribute

`change_name`(*col_name*, *new_name*)

Change a `Column`'s name.

Parameters: **col_name** : *str*

The current name of the column

new_name : *str*

The new name of the column

`change_unit`(*col_name*, *new_unit*)

Change a `Column`'s unit.

Parameters: **col_name** : *str or int*

The column name or index

new_unit : *str*

The new unit for the column

`del_col`(*col_name*)

Delete (the definition of) one `Column`.

col_name : *str or int*

The column's name or index

`info` (*attrib='all', output=None*)

Get attribute(s) information of the column definition.

Parameters:

attrib : *str*

Can be one or more of the attributes listed in `astropy.io.fits.column.KEYWORD_ATTRIBUTES`. The default is `"all"` which will print out all attributes. It forgives plurals and blanks. If there are two or more attribute names, they must be separated by comma(s).

output : *file, optional*

File-like object to output to. Outputs to stdout by default. If **`False`**, returns the attributes as a **`dict`** instead.

Notes

This function doesn't return anything by default; it just prints to stdout.

FITS_rec

class `astropy.io.fits.` **`FITS_rec`** (*input*)

Bases: **`numpy.recarray`**

FITS record array class.

`FITS_rec` is the data part of a table HDU's data part. This is a layer over the **`recarray`**, so we can deal with scaled columns.

It inherits all of the standard methods from **`numpy.ndarray`**.

Construct a FITS record array from a recarray.

property **`columns`**

A user-visible accessor for the coldefs.

`copy` (*order='C'*)

The Numpy documentation lies; **`numpy.ndarray.copy`** is not equivalent to **`numpy.copy`**. Differences include that it re-views the copied array as self's ndarray subclass, as though it were taking a slice; this means `__array_finalize__` is called and the copy shares all the array attributes (including `._converted`!). So we need to make a deep copy of all those attributes so that the two arrays truly do not share any data.

`field` (*key*)

A view of a `Column`'s data as an array.

property **`formats`**

List of column FITS formats.

classmethod **`from_columns`**(*columns*, *nrows=0*, *fill=False*, *character_as_bytes=False*)

Given a `ColDefs` object of unknown origin, initialize a new `FITS_rec` object.

Note

This was originally part of the `new_table` function in the table module but was moved into a class method since most of its functionality always had more to do with initializing a `FITS_rec` object than anything else, and much of it also overlapped with `FITS_rec._scale_back`.

Parameters:

columns : *sequence of* `Column` *or a* `ColDefs`

The columns from which to create the table data. If these columns have data arrays attached that data may be used in initializing the new table. Otherwise the input columns will be used as a template for a new table with the requested number of rows.

nrows : *int*

Number of rows in the new table. If the input columns have data associated with them, the size of the largest input column is used. Otherwise the default is 0.

fill : *bool*

If `True`, will fill all cells with zeros or blanks. If `False`, copy the data from input, undefined cells will still be filled with zeros/blanks.

property **`names`**

List of column names.

FITS_record

class `astropy.io.fits.`**`FITS_record`**(*input*, *row=0*, *start=None*, *end=None*, *step=None*, *base=None*, ***kwargs*)

Bases: `object`

FITS record class.

`FITS_record` is used to access records of the `FITS_rec` object. This will allow us to deal with scaled columns. It also handles conversion/scaling of columns in ASCII tables. The `FITS_record` class expects a `FITS_rec` object as input.

Parameters: **input** : *array*

The array to wrap.

row : *int, optional*

The starting logical row of the array.

start : *int, optional*

The starting column in the row associated with this object. Used for subsetting the columns of the `FITS_rec` object.

end : *int, optional*

The ending column in the row associated with this object. Used for subsetting the columns of the `FITS_rec` object.

`field` (*field*)

Get the field data of the record.

`setfield` (*field*, *value*)

Set the field data of the record.

Table Functions

`tabledump()`

`astropy.io.fits.`**`tabledump`**(*filename*, *datafile=None*, *cdfile=None*, *hfile=None*, *ext=1*, *overwrite=False*)

Dump a table HDU to a file in ASCII format. The table may be dumped in three separate files, one containing column definitions, one containing header parameters, and one for table data.

Parameters: **filename** : *file path, file object or file-like object*

Input fits file.

datafile : *file path, file object or file-like object, optional*

Output data file. The default is the root name of the input fits file appended with an underscore, followed by the extension number (ext), followed by the extension `.txt`.

cdfile : *file path, file object or file-like object, optional*

Output column definitions file. The default is `None`, no column definitions output is produced.

hfile : *file path, file object or file-like object, optional*

Output header parameters file. The default is `None`, no header parameters output is produced.

ext : *int*

The number of the extension containing the table HDU to be dumped.

overwrite : *bool, optional*

If `True`, overwrite the output file if it exists. Raises an `OSError` if `False` and the output file exists. Default is `False`.

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

Notes

The primary use for the `tabledump` function is to allow editing in a standard text editor of the table data and parameters. The `tableload` function can be used to reassemble the table from the three ASCII files.

- **datafile:** Each line of the data file represents one row of table data. The data is output one column at a time in column order. If a column contains an array, each element of the column array in the current row is output before moving on to the next column. Each row ends with a new line.

  Integer data is output right-justified in a 21-character field followed by a blank. Floating point data is output right justified using 'g' format in a 21-character field with 15 digits of precision, followed by a blank. String data that does not contain whitespace is output left-justified in a field whose width matches the width specified in the `TFORM` header parameter for the column, followed by a blank. When the string data contains whitespace characters, the string is enclosed in quotation marks (`""`). For the last data element in a row, the trailing blank in the field is replaced by a new line character.

For column data containing variable length arrays (‘P’ format), the array data is preceded by the string `'VLA_Length= '` and the integer length of the array for that row, left-justified in a 21-character field, followed by a blank.

Note

This format does *not* support variable length arrays using the (‘Q’ format) due to difficult to overcome ambiguities. What this means is that this file format cannot support VLA columns in tables stored in files that are over 2 GB in size.

For column data representing a bit field (‘X’ format), each bit value in the field is output right-justified in a 21-character field as 1 (for true) or 0 (for false).

- **cdfile:** Each line of the column definitions file provides the definitions for one column in the table. The line is broken up into 8, sixteen-character fields. The first field provides the column name (`TTYPEn`). The second field provides the column format (`TFORMn`). The third field provides the display format (`TDISPn`). The fourth field provides the physical units (`TUNITn`). The fifth field provides the dimensions for a multidimensional array (`TDIMn`). The sixth field provides the value that signifies an undefined value (`TNULLn`). The seventh field provides the scale factor (`TSCALn`). The eighth field provides the offset value (`TZEROn`). A field value of `""` is used to represent the case where no value is provided.
- **hfile:** Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

tableload()

`astropy.io.fits.` **`tableload`** (*datafile*, *cdfile*, *hfile=None*)

Create a table from the input ASCII files. The input is from up to three separate files, one containing column definitions, one containing header parameters, and one containing column data. The header parameters file is not required. When the header parameters file is absent a minimal header is constructed.

Parameters: **datafile** : *file path, file object or file-like object*

Input data file containing the table data in ASCII format.

cdfile : *file path, file object or file-like object*

Input column definition file containing the names, formats, display formats, physical units, multidimensional array dimensions, undefined values, scale factors, and offsets associated with the columns in the table.

hfile : *file path, file object or file-like object, optional*

Input parameter definition file containing the header parameter definitions to be associated with the table. If `None`, a minimal header is constructed.

Notes

The primary use for the `tableload` function is to allow the input of ASCII data that was edited in a standard text editor of the table data and parameters. The tabledump function can be used to create the initial ASCII files.

- **datafile:** Each line of the data file represents one row of table data. The data is output one column at a time in column order. If a column contains an array, each element of the column array in the current row is output before moving on to the next column. Each row ends with a new line.

  Integer data is output right-justified in a 21-character field followed by a blank. Floating point data is output right justified using 'g' format in a 21-character field with 15 digits of precision, followed by a blank. String data that does not contain whitespace is output left-justified in a field whose width matches the width specified in the `TFORM` header parameter for the column, followed by a blank. When the string data contains whitespace characters, the string is enclosed in quotation marks (`""`). For the last data element in a row, the trailing blank in the field is replaced by a new line character.

  For column data containing variable length arrays ('P' format), the array data is preceded by the string `'VLA_Length= '` and the integer length of the array for that row, left-justified in a 21-character field, followed by a blank.

  Note

  This format does *not* support variable length arrays using the ('Q' format) due to difficult to overcome ambiguities. What this means is that this file format cannot support VLA columns in tables stored in files that are over 2 GB in size.

For column data representing a bit field ('X' format), each bit value in the field is output right-justified in a 21-character field as 1 (for true) or 0 (for false).

- **cdfile:** Each line of the column definitions file provides the definitions for one column in the table. The line is broken up into 8, sixteen-character fields. The first field provides the column name (`TTYPEn`). The second field provides the column format (`TFORMn`). The third field provides the display format (`TDISPn`). The fourth field provides the physical units (`TUNITn`). The fifth field provides the dimensions for a multidimensional array (`TDIMn`). The sixth field provides the value that signifies an undefined value (`TNULLn`). The seventh field provides the scale factor (`TSCALn`). The eighth field provides the offset value (`TZEROn`). A field value of `""` is used to represent the case where no value is provided.
- **hfile:** Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

`table_to_hdu()`

`astropy.io.fits.` **`table_to_hdu`**(*table*, *character_as_bytes=False*)

Convert an `Table` object to a FITS `BinTableHDU`.

Parameters: **table** : *astropy.table.Table*

The table to convert.

character_as_bytes : *bool*

Whether to return bytes for string columns when accessed from the HDU. By default this is `False` and (unicode) strings are returned, but for large tables this may use up a lot of memory.

Returns: **table_hdu** : *`BinTableHDU`*

The FITS binary table HDU.

Images

`ImageHDU`

class `astropy.io.fits.` **`ImageHDU`**(*data=None*, *header=None*, *name=None*, *do_not_scale_image_data=False*, *uint=True*, *scale_back=None*, *ver=None*)

Bases: **`astropy.io.fits.hdu.image._ImageBaseHDU`**, **`astropy.io.fits.hdu.base.ExtensionHDU`**

FITS image extension HDU class.

Construct an image HDU.

Parameters: **data** : *array*

The data in the HDU.

header : ***Header***

The header to be used (as a template). If `header` is **`None`**, a minimal header will be provided.

name : *str, optional*

The name of the HDU, will be the value of the keyword `EXTNAME`.

do_not_scale_image_data : *bool, optional*

If **`True`**, image data is not scaled using BSCALE/BZERO values when read. (default: False)

uint : *bool, optional*

Interpret signed integer data where `BZERO` is the central value and `BSCALE == 1` as unsigned integer data. For example, `int16` data with `BZERO = 32768` and `BSCALE = 1` would be treated as `uint16` data. (default: True)

scale_back : *bool, optional*

If **`True`**, when saving changes to a file that contained scaled image data, restore the data to the original type and reapply the original BSCALE/BZERO values. This could lead to loss of accuracy if scaling back to integer values after performing floating point operations on the data. Pseudo-unsigned integers are automatically rescaled unless scale_back is explicitly set to **`False`**. (default: None)

ver : *int > 0 or None, optional*

The ver of the HDU, will be the value of the keyword `EXTVER`. If not given or None, it defaults to the value of the `EXTVER` card of the `header` or 1. (default: None)

`add_checksum`(*when=None*, *override_datasum=False*, *checksum_keyword='CHECKSUM'*, *datasum_keyword='DATASUM'*)

Add the `CHECKSUM` and `DATASUM` cards to this HDU with the values set to the checksum calculated for the HDU and the data respectively. The addition of the `DATASUM` card may be overridden.

Parameters: **when** : *str, optional*

comment string for the cards; by default the comments will represent the time when the checksum was calculated

override_datasum : *bool, optional*

add the `CHECKSUM` card only

checksum_keyword : *str, optional*

The name of the header keyword to store the checksum value in; this is typically 'CHECKSUM' per convention, but there exist use cases in which a different keyword should be used

datasum_keyword : *str, optional*

See `checksum_keyword`

Notes

For testing purposes, first call **`add_datasum`** with a `when` argument, then call **`add_checksum`** with a `when` argument and `override_datasum` set to **`True`**. This will provide consistent comments for both cards and enable the generation of a `CHECKSUM` card with a consistent value.

`add_datasum` (*when=None*, *datasum_keyword='DATASUM'*)

Add the `DATASUM` card to this HDU with the value set to the checksum calculated for the data.

Parameters: **when** : *str, optional*

Comment string for the card that by default represents the time when the checksum was calculated

datasum_keyword : *str, optional*

The name of the header keyword to store the datasum value in; this is typically 'DATASUM' per convention, but there exist use cases in which a different keyword should be used

Returns: **checksum** : *int*

The calculated datasum

Notes

For testing purposes, provide a `when` argument to enable the comment value in the card to remain consistent. This will enable the generation of a `CHECKSUM` card with a consistent value.

`copy` ()

Make a copy of the HDU, both header and data are copied.

property `data`

Image/array data as a `ndarray`.

Please remember that the order of axes on an Numpy array are opposite of the order specified in the FITS file. For example for a 2D image the “rows” or y-axis are the first dimension, and the “columns” or x-axis are the second dimension.

If the data is scaled using the BZERO and BSCALE parameters, this attribute returns the data scaled to its physical values unless the file was opened with `do_not_scale_image_data=True`.

`filebytes` ()

Calculates and returns the number of bytes that this HDU will write to a file.

`fileinfo` ()

Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the `HDUList`.

Returns: dict or None

The dictionary details information about the locations of this HDU within an associated file. Returns `None` when the HDU is not associated with a file.
Dictionary contents:

Key	Value
file	File object associated with the HDU
filemode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
hdrLoc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

classmethod `fromstring` (*data*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Creates a new HDU object of the appropriate type from a string containing the HDU’s entire header and, optionally, its data.

Note: When creating a new HDU from a string without a backing file object, the data of that HDU may be read-only. It depends on whether the underlying string was an immutable Python str/bytes object, or some kind of read-write memory buffer such as a `memoryview`.

Parameters: **data** : *str, bytearray, memoryview, ndarray*

A byte string containing the HDU's header and data.

checksum : *bool, optional*

Check the HDU's checksum and/or datasum.

ignore_missing_end : *bool, optional*

Ignore a missing end card in the header data. Note that without the end card the end of the header may be ambiguous and resulted in a corrupt HDU. In this case the assumption is that the first 2880 block that does not begin with valid FITS header data is the beginning of the data.

kwargs : *optional*

May consist of additional keyword arguments specific to an HDU type–these correspond to keywords recognized by the constructors of different HDU classes such as `PrimaryHDU`, `ImageHDU`, or `BinTableHDU`. Any unrecognized keyword arguments are simply ignored.

classmethod `match_header`(*header*)

_ImageBaseHDU is sort of an abstract class for HDUs containing image data (as opposed to table data) and should never be used directly.

classmethod `readfrom`(*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with `open()` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto()`.

Parameters: **fileobj** : *file object or file-like object*

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : *bool*

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : *bool*

Do not issue an exception when opening a file that is missing an `END` card in the last header.

`req_cards`(*keyword*, *pos*, *test*, *fix_value*, *option*, *errlist*)

Check the existence, location, and value of a required `Card`.

Parameters: **keyword** : *str*

The keyword to validate

pos : *int, callable*

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument–the actual position of the keyword–and return **`True`** or **`False`**. This can be used for custom evaluation. For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : *callable*

This should be a callable (generally a function) that is passed the value of the given keyword and returns **`True`** or **`False`**. This can be used to validate the value associated with the given keyword.

fix_value : *str, int, float, complex, bool, None*

A valid value for a FITS keyword to to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If **`None`**, there is no replacement value and the keyword is unfixable.

option : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `+warn`, or `+exception" (e.g. ``"fix+warn"`). See Verification Options for more info.

errlist : *list*

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to **`req_cards`**.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

`run_option` (*option='warn'*, *err_text=''*, *fix_text='Fixed.'*, *fix=None*, *fixable=True*)

Execute the verification with selected option.

`scale` (*type=None*, *option='old'*, *bscale=None*, *bzero=None*)

Scale image data by using `BSCALE` / `BZERO`.

Call to this method will scale **`data`** and update the keywords of `BSCALE` and `BZERO` in the HDU's header. This method should only be used right before writing to the output file, as the data will be scaled and is therefore not very usable after the call.

Parameters: **type** : *str, optional*

destination data type, use a string representing a numpy dtype name, (e.g. `'uint8'`, `'int16'`, `'float32'` etc.). If is **`None`**, use the current data type.

option : *str, optional*

How to scale the data: `"old"` uses the original `BSCALE` and `BZERO` values from when the data was read/created (defaulting to 1 and 0 if they don't exist). For integer data only, `"minmax"` uses the minimum and maximum of the data to scale. User-specified `bscale` / `bzero` values always take precedence.

bscale, bzero : *int, optional*

User-specified `BSCALE` and `BZERO` values

property **`section`**

Access a section of the image array without loading the entire array into memory. The **`Section`** object returned by this attribute is not meant to be used directly by itself. Rather, slices of the section return the appropriate slice of the data, and loads *only* that section into memory.

Sections are mostly obsoleted by memmap support, but should still be used to deal with very large scaled images. See the Data Sections section of the Astropy documentation for more details.

property **`shape`**

Shape of the image array–should be equivalent to `self.data.shape`.

property **`size`**

Size (in bytes) of the data portion of the HDU.

`update_header` ()

Update the header keywords to agree with the data.

verify (*option='warn'*)

Verify all values in the instance.

Parameters: **option** : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `"+warn"`, or `"+exception"` (e.g. `"fix+warn"`). See Verification Options for more info.

verify_checksum ()

Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU CHECKSUM.

Returns: **valid** : *int*

- 0 - failure
- 1 - success
- 2 - no `CHECKSUM` keyword present

verify_datasum ()

Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

Returns: **valid** : *int*

- 0 - failure
- 1 - success
- 2 - no `DATASUM` keyword present

writeto (*name*, *output_verify='exception'*, *overwrite=False*, *checksum=False*)

Works similarly to the normal writeto(), but prepends a default **`PrimaryHDU`** are required by extension HDUs (which cannot stand on their own).

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

CompImageHDU

class `astropy.io.fits.` **CompImageHDU** (*data=None*, *header=None*, *name=None*, *compression_type='RICE_1'*, *tile_size=None*, *hcomp_scale=0*, *hcomp_smooth=0*, *quantize_level=16.0*, *quantize_method=- 1*, *dither_seed=0*, *do_not_scale_image_data=False*, *uint=False*, *scale_back=False*, ***kwargs*)

Bases: `astropy.io.fits.BinTableHDU`

Compressed Image HDU class.

Parameters:

data : *array, optional*

Uncompressed image data

header : *`Header`, optional*

Header to be associated with the image; when reading the HDU from a file (data=DELAYED), the header read from the file

name : *str, optional*

The `EXTNAME` value; if this value is `None`, then the name from the input image header will be used; if there is no name in the input image header then the default name `COMPRESSED_IMAGE` is used.

compression_type : *str, optional*

Compression algorithm: one of `'RICE_1'`, `'RICE_ONE'`, `'PLIO_1'`, `'GZIP_1'`, `'GZIP_2'`, `'HCOMPRESS_1'`

tile_size : *int, optional*

Compression tile sizes. Default treats each row of image as a tile.

hcomp_scale : *float, optional*

HCOMPRESS scale parameter

hcomp_smooth : *float, optional*

HCOMPRESS smooth parameter

quantize_level : *float, optional*

Floating point quantization level; see note below

quantize_method : *int, optional*

Floating point quantization dithering method; can be either `NO_DITHER` (-1; default), `SUBTRACTIVE_DITHER_1` (1), or `SUBTRACTIVE_DITHER_2` (2); see note below

dither_seed : *int, optional*

Random seed to use for dithering; can be either an integer in the range 1 to 1000 (inclusive), `DITHER_SEED_CLOCK` (0; default), or `DITHER_SEED_CHECKSUM` (-1); see note below

Notes

The astropy.io.fits package supports 2 methods of image compression:

1. The entire FITS file may be externally compressed with the gzip or pkzip utility programs, producing a `*.gz` or `*.zip` file, respectively. When reading compressed files of this type, Astropy first uncompresses the entire file into a temporary file before performing the requested read operations. The astropy.io.fits package does not support writing to these types of compressed files. This type of compression is supported in the `_File` class, not in the **`CompImageHDU`** class. The file compression type is recognized by the `.gz` or `.zip` file name extension.
2. The **`CompImageHDU`** class supports the FITS tiled image compression convention in which the image is subdivided into a grid of rectangular tiles, and each tile of pixels is individually compressed. The details of this FITS compression convention are described at the FITS Support Office web site. Basically, the compressed image tiles are stored in rows of a variable length array column in a FITS binary table. The astropy.io.fits recognizes that this binary table extension contains an image and treats it as if it were an image extension. Under this tile-compression format, FITS header keywords remain uncompressed. At this time, Astropy does not support the ability to extract and uncompress sections of the image without having to uncompress the entire image.

The astropy.io.fits package supports 3 general-purpose compression algorithms plus one other special-purpose compression technique that is designed for data masks with positive integer pixel values. The 3 general purpose algorithms are GZIP, Rice, and HCOMPRESS, and the special-purpose technique is the IRAF pixel list compression technique (PLIO). The `compression_type` parameter defines the compression algorithm to be used.

The FITS image can be subdivided into any desired rectangular grid of compression tiles. With the GZIP, Rice, and PLIO algorithms, the default is to take each row of the image as a tile. The HCOMPRESS algorithm is inherently 2-dimensional in nature, so the default in this case is to take 16 rows of the image per tile. In most cases, it makes little difference what tiling pattern is used, so the default tiles are usually adequate. In the case of very small images, it could be more efficient to compress the whole image as a single tile. Note that the image dimensions are not required to be an integer multiple of the tile dimensions; if not, then the tiles at the edges of the image will be smaller than the other tiles. The `tile_size` parameter may be provided as a list of tile sizes, one for each dimension in the image. For example a `tile_size` value of `[100,100]` would divide a 300 X 300 image into 9 100 X 100 tiles.

The 4 supported image compression algorithms are all ‘lossless’ when applied to integer FITS images; the pixel values are preserved exactly with no loss of information during the compression and uncompression process. In addition, the HCOMPRESS algorithm supports a ‘lossy’ compression mode that will produce larger amount of image compression. This is achieved by specifying a non-zero value for the `hcomp_scale` parameter. Since the amount of compression that is achieved depends directly on the RMS noise in the image, it is usually more convenient to specify the `hcomp_scale` factor relative to the RMS noise. Setting `hcomp_scale = 2.5` means use a scale factor that is 2.5 times the calculated RMS noise in the image tile. In some cases it may be desirable to specify the exact scaling to be used, instead of specifying it relative to the calculated noise value. This may be done by specifying the negative of the desired scale value (typically in the range -2 to -100).

Very high compression factors (of 100 or more) can be achieved by using large `hcomp_scale` values, however, this can produce undesirable ‘blocky’ artifacts in the compressed image. A variation of the HCOMPRESS algorithm (called HSCOMPRESS) can be used in this case to apply a small amount of smoothing of the image when it is uncompressed to help cover up these artifacts. This smoothing is purely cosmetic and does not cause any significant change to the image pixel values. Setting the `hcomp_smooth` parameter to 1 will engage the smoothing algorithm.

Floating point FITS images (which have `BITPIX` = -32 or -64) usually contain too much ‘noise’ in the least significant bits of the mantissa of the pixel values to be effectively compressed with any lossless algorithm. Consequently, floating point images are first quantized into scaled integer pixel values (and thus throwing away much of the noise) before being compressed with the specified algorithm (either GZIP, RICE, or HCOMPRESS). This technique produces much higher compression factors than simply using the GZIP utility to externally compress the whole FITS file, but it also means that the original floating point value pixel values are not exactly preserved. When done properly, this integer scaling technique will only discard the insignificant noise while still preserving all the real information in the image. The amount of precision that is retained in the pixel values is controlled by the `quantize_level` parameter. Larger values will result in compressed images whose pixels more closely match the floating point pixel values, but at the same time the amount of compression that is achieved will be reduced. Users should experiment with different values for this parameter to determine the optimal value that preserves all the useful information in the image, without needlessly preserving all the ‘noise’ which will hurt the compression efficiency.

The default value for the `quantize_level` scale factor is 16, which means that scaled integer pixel values will be quantized such that the difference between adjacent integer values will be 1/16th of the noise level in the image.

background. An optimized algorithm is used to accurately estimate the noise in the image. As an example, if the RMS noise in the background pixels of an image = 32.0, then the spacing between adjacent scaled integer pixel values will equal 2.0 by default. Note that the RMS noise is independently calculated for each tile of the image, so the resulting integer scaling factor may fluctuate slightly for each tile. In some cases, it may be desirable to specify the exact quantization level to be used, instead of specifying it relative to the calculated noise value. This may be done by specifying the negative of desired quantization level for the value of `quantize_level`. In the previous example, one could specify `quantize_level = -2.0` so that the quantized integer levels differ by 2.0. Larger negative values for `quantize_level` means that the levels are more coarsely-spaced, and will produce higher compression factors.

The quantization algorithm can also apply one of two random dithering methods in order to reduce bias in the measured intensity of background regions. The default method, specified with the constant `SUBTRACTIVE_DITHER_1` adds dithering to the zero-point of the quantization array itself rather than adding noise to the actual image. The random noise is added on a pixel-by-pixel basis, so in order restore each pixel from its integer value to its floating point value it is necessary to replay the same sequence of random numbers for each pixel (see below). The other method, `SUBTRACTIVE_DITHER_2`, is exactly like the first except that before dithering any pixel with a floating point value of `0.0` is replaced with the special integer value `-2147483647`. When the image is uncompressed, pixels with this value are restored back to `0.0` exactly. Finally, a value of `NO_DITHER` disables dithering entirely.

As mentioned above, when using the subtractive dithering algorithm it is necessary to be able to generate a (pseudo-)random sequence of noise for each pixel, and replay that same sequence upon decompressing. To facilitate this, a random seed between 1 and 10000 (inclusive) is used to seed a random number generator, and that seed is stored in the `ZDITHER0` keyword in the header of the compressed HDU. In order to use that seed to generate the same sequence of random numbers the same random number generator must be used at compression and decompression time; for that reason the tiled image convention provides an implementation of a very simple pseudo-random number generator. The seed itself can be provided in one of three ways, controllable by the `dither_seed` argument: It may be specified manually, or it may be generated arbitrarily based on the system's clock (`DITHER_SEED_CLOCK`) or based on a checksum of the pixels in the image's first tile (`DITHER_SEED_CHECKSUM`). The clock-based method is the default, and is sufficient to ensure that the value is reasonably “arbitrary” and that the same seed is unlikely to be generated sequentially. The checksum method, on the other hand, ensures that the same seed is used every time for a specific image. This is particularly useful for software testing as it ensures that the same image will always use the same seed.

`add_checksum`(*when=None*, *override_datasum=False*, *checksum_keyword='CHECKSUM'*, *datasum_keyword='DATASUM'*)

Add the `CHECKSUM` and `DATASUM` cards to this HDU with the values set to the checksum calculated for the HDU and the data respectively. The addition of the `DATASUM` card may be overridden.

Parameters: **when** : *str, optional*

comment string for the cards; by default the comments will represent the time when the checksum was calculated

override_datasum : *bool, optional*

add the `CHECKSUM` card only

checksum_keyword : *str, optional*

The name of the header keyword to store the checksum value in; this is typically 'CHECKSUM' per convention, but there exist use cases in which a different keyword should be used

datasum_keyword : *str, optional*

See `checksum_keyword`

Notes

For testing purposes, first call **`add_datasum`** with a `when` argument, then call **`add_checksum`** with a `when` argument and `override_datasum` set to **`True`**. This will provide consistent comments for both cards and enable the generation of a `CHECKSUM` card with a consistent value.

`add_datasum`(*when=None*, *datasum_keyword='DATASUM'*)

Add the `DATASUM` card to this HDU with the value set to the checksum calculated for the data.

Parameters: **when** : *str, optional*

Comment string for the card that by default represents the time when the checksum was calculated

datasum_keyword : *str, optional*

The name of the header keyword to store the datasum value in; this is typically 'DATASUM' per convention, but there exist use cases in which a different keyword should be used

Returns: **checksum** : *int*

The calculated datasum

Notes

For testing purposes, provide a `when` argument to enable the comment value in the card to remain consistent. This will enable the generation of a `CHECKSUM` card with a consistent value.

property **`columns`**

The **`ColDefs`** objects describing the columns in this table.

`copy` ()

Make a copy of the table HDU, both header and data are copied.

`dump` (*datafile=None*, *cdfile=None*, *hfile=None*, *overwrite=False*)

Dump the table HDU to a file in ASCII format. The table may be dumped in three separate files, one containing column definitions, one containing header parameters, and one for table data.

Parameters: **datafile** : *file path, file object or file-like object, optional*

Output data file. The default is the root name of the fits file associated with this HDU appended with the extension `.txt`.

cdfile : *file path, file object or file-like object, optional*

Output column definitions file. The default is **`None`**, no column definitions output is produced.

hfile : *file path, file object or file-like object, optional*

Output header parameters file. The default is **`None`**, no header parameters output is produced.

overwrite : *bool, optional*

If `True`, overwrite the output file if it exists. Raises an `OSError` if `False` and the output file exists. Default is `False`.

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

Notes

The primary use for the **`dump`** method is to allow viewing and editing the table data and parameters in a standard text editor. The **`load`** method can be used to create a new table from the three plain text (ASCII) files.

- **datafile:** Each line of the data file represents one row of table data. The data is output one column at a time in column order. If a column contains an array, each element of the column array in the current row is output before moving on to the next column. Each row ends with a new line.

  Integer data is output right-justified in a 21-character field followed by a blank. Floating point data is output right justified using 'g' format in a 21-character field with 15 digits of precision, followed by a blank. String data that does not contain whitespace is output left-justified in a field whose width matches the width specified in the `TFORM` header parameter for the column, followed by a blank. When the string data contains whitespace characters, the string is enclosed in quotation marks (`""`). For the last data element in a row, the trailing blank in the field is replaced by a new line character.

  For column data containing variable length arrays ('P' format), the array data is preceded by the string `'VLA_Length= '` and the integer length of the array for that row, left-justified in a 21-character field, followed by a blank.

  Note

  This format does *not* support variable length arrays using the ('Q' format) due to difficult to overcome ambiguities. What this means is that this file format cannot support VLA columns in tables stored in files that are over 2 GB in size.

  For column data representing a bit field ('X' format), each bit value in the field is output right-justified in a 21-character field as 1 (for true) or 0 (for false).

- **cdfile:** Each line of the column definitions file provides the definitions for one column in the table. The line is broken up into 8, sixteen-character fields. The first field provides the column name (`TTYPEn`). The second field provides the column format (`TFORMn`). The third field provides the display format (`TDISPn`). The fourth field provides the physical units (`TUNITn`). The fifth field provides the dimensions for a multidimensional array (`TDIMn`). The sixth field provides the value that signifies an undefined value (`TNULLn`). The seventh field provides the scale factor (`TSCALn`). The eighth field provides the offset value (`TZEROn`). A field value of `""` is used to represent the case where no value is provided.

- **hfile:** Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

`filebytes` ()

Calculates and returns the number of bytes that this HDU will write to a file.

`fileinfo` ()

Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the **`HDUList`**.

Returns: dict or None

The dictionary details information about the locations of this HDU within an associated file. Returns **`None`** when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
filemode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
hdrLoc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

classmethod **`from_columns`** (*columns*, *header=None*, *nrows=0*, *fill=False*, *character_as_bytes=False*, ***kwargs*)

Given either a **`ColDefs`** object, a sequence of **`Column`** objects, or another table HDU or table data (a **`FITS_rec`** or multi-field **`numpy.ndarray`** or **`numpy.recarray`** object, return a new table HDU of the class this method was called on using the column definition from the input.

See also **`FITS_rec.from_columns`**.

Parameters: **columns** : *sequence of* ***`Column`***, ***`ColDefs`***, *or other*

The columns from which to create the table data, or an object with a column-like structure from which a **`ColDefs`** can be instantiated. This includes an existing **`BinTableHDU`** or **`TableHDU`**, or a **`numpy.recarray`** to give some examples.

If these columns have data arrays attached that data may be used in initializing the new table. Otherwise the input columns will be used as a template for a new table with the requested number of rows.

header : ***`Header`***

An optional **`Header`** object to instantiate the new HDU yet. Header keywords specifically related to defining the table structure (such as the "TXXXn" keywords like TTYPEn) will be overridden by the supplied column definitions, but all other informational and data model-specific keywords are kept.

nrows : *int*

Number of rows in the new table. If the input columns have data associated with them, the size of the largest input column is used. Otherwise the default is 0.

fill : *bool*

If **`True`**, will fill all cells with zeros or blanks. If **`False`**, copy the data from input, undefined cells will still be filled with zeros/blanks.

character_as_bytes : *bool*

Whether to return bytes for string columns when accessed from the HDU. By default this is **`False`** and (unicode) strings are returned, but for large tables this may use up a lot of memory.

Notes

Any additional keyword arguments accepted by the HDU class's `__init__` may also be passed in as keyword arguments.

classmethod **`fromstring`**(*data*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Creates a new HDU object of the appropriate type from a string containing the HDU's entire header and, optionally, its data.

Note: When creating a new HDU from a string without a backing file object, the data of that HDU may be read-only. It depends on whether the underlying string was an immutable Python str/bytes object, or some kind of read-write

memory buffer such as a **memoryview**.

Parameters: **data** : *str, bytearray, memoryview, ndarray*

A byte string containing the HDU's header and data.

checksum : *bool, optional*

Check the HDU's checksum and/or datasum.

ignore_missing_end : *bool, optional*

Ignore a missing end card in the header data. Note that without the end card the end of the header may be ambiguous and resulted in a corrupt HDU. In this case the assumption is that the first 2880 block that does not begin with valid FITS header data is the beginning of the data.

kwargs : *optional*

May consist of additional keyword arguments specific to an HDU type–these correspond to keywords recognized by the constructors of different HDU classes such as **PrimaryHDU**, **ImageHDU**, or **BinTableHDU**. Any unrecognized keyword arguments are simply ignored.

classmethod **load**(*datafile*, *cdfile=None*, *hfile=None*, *replace=False*, *header=None*)

Create a table from the input ASCII files. The input is from up to three separate files, one containing column definitions, one containing header parameters, and one containing column data.

The column definition and header parameters files are not required. When absent the column definitions and/or header parameters are taken from the header object given in the header argument; otherwise sensible defaults are inferred (though this mode is not recommended).

Parameters: **datafile** : *file path, file object or file-like object*

Input data file containing the table data in ASCII format.

cdfile : *file path, file object, file-like object, optional*

Input column definition file containing the names, formats, display formats, physical units, multidimensional array dimensions, undefined values, scale factors, and offsets associated with the columns in the table. If `None`, the column definitions are taken from the current values in this object.

hfile : *file path, file object, file-like object, optional*

Input parameter definition file containing the header parameter definitions to be associated with the table. If `None`, the header parameter definitions are taken from the current values in this objects header.

replace : *bool, optional*

When `True`, indicates that the entire header should be replaced with the contents of the ASCII file instead of just updating the current header.

header : *`Header`, optional*

When the cdfile and hfile are missing, use this Header object in the creation of the new table and HDU. Otherwise this Header supersedes the keywords from hfile, which is only used to update values not present in this Header, unless `replace=True` in which this Header's values are completely replaced with the values from hfile.

Notes

The primary use for the `load` method is to allow the input of ASCII data that was edited in a standard text editor of the table data and parameters. The `dump` method can be used to create the initial ASCII files.

- **datafile:** Each line of the data file represents one row of table data. The data is output one column at a time in column order. If a column contains an array, each element of the column array in the current row is output before moving on to the next column. Each row ends with a new line.

  Integer data is output right-justified in a 21-character field followed by a blank. Floating point data is output right justified using 'g' format in a 21-character field with 15 digits of precision, followed by a blank. String data that does not contain whitespace is output left-justified in a field whose width matches the width specified in the `TFORM` header parameter for the column, followed by a blank. When the string data contains whitespace characters, the string is enclosed in quotation marks (`""`). For the last data element in a row, the trailing blank in the field is replaced by a new line character.

For column data containing variable length arrays ('P' format), the array data is preceded by the string `'VLA_Length= '` and the integer length of the array for that row, left-justified in a 21-character field, followed by a blank.

Note

This format does *not* support variable length arrays using the ('Q' format) due to difficult to overcome ambiguities. What this means is that this file format cannot support VLA columns in tables stored in files that are over 2 GB in size.

For column data representing a bit field ('X' format), each bit value in the field is output right-justified in a 21-character field as 1 (for true) or 0 (for false).

- **cdfile:** Each line of the column definitions file provides the definitions for one column in the table. The line is broken up into 8, sixteen-character fields. The first field provides the column name (`TTYPEn`). The second field provides the column format (`TFORMn`). The third field provides the display format (`TDISPn`). The fourth field provides the physical units (`TUNITn`). The fifth field provides the dimensions for a multidimensional array (`TDIMn`). The sixth field provides the value that signifies an undefined value (`TNULLn`). The seventh field provides the scale factor (`TSCALn`). The eighth field provides the offset value (`TZEROn`). A field value of `""` is used to represent the case where no value is provided.
- **hfile:** Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

classmethod **`match_header`** (*header*)

This is an abstract type that implements the shared functionality of the ASCII and Binary Table HDU types, which should be used instead of this.

classmethod **`readfrom`** (*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with `open()` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto()`.

Parameters:

fileobj : *file object or file-like object*

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : *bool*

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : *bool*

Do not issue an exception when opening a file that is missing an `END` card in the last header.

`req_cards` (*keyword*, *pos*, *test*, *fix_value*, *option*, *errlist*)

Check the existence, location, and value of a required `Card`.

Parameters: **keyword** : *str*

The keyword to validate

pos : *int, callable*

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument–the actual position of the keyword–and return **`True`** or **`False`**. This can be used for custom evaluation. For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : *callable*

This should be a callable (generally a function) that is passed the value of the given keyword and returns **`True`** or **`False`**. This can be used to validate the value associated with the given keyword.

fix_value : *str, int, float, complex, bool, None*

A valid value for a FITS keyword to to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If **`None`**, there is no replacement value and the keyword is unfixable.

option : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `+warn`, or `+exception" (e.g. ``"fix+warn"`). See Verification Options for more info.

errlist : *list*

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to **`req_cards`**.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

`run_option`(*option='warn'*, *err_text=''*, *fix_text='Fixed.'*, *fix=None*, *fixable=True*)

Execute the verification with selected option.

`scale`(*type=None*, *option='old'*, *bscale=1*, *bzero=0*)

Scale image data by using `BSCALE` and `BZERO`.

Calling this method will scale `self.data` and update the keywords of `BSCALE` and `BZERO` in `self._header` and `self._image_header`. This method should only be used right before writing to the output file, as the data will be scaled and is therefore not very usable after the call.

Parameters: **type** : *str, optional*

destination data type, use a string representing a numpy dtype name, (e.g. `'uint8'`, `'int16'`, `'float32'` etc.). If is `None`, use the current data type.

option : *str, optional*

how to scale the data: if `"old"`, use the original `BSCALE` and `BZERO` values when the data was read/created. If `"minmax"`, use the minimum and maximum of the data to scale. The option will be overwritten by any user-specified bscale/bzero values.

bscale, bzero : *int, optional*

user specified `BSCALE` and `BZERO` values.

property `shape`

Shape of the image array–should be equivalent to `self.data.shape`.

property `size`

Size (in bytes) of the data portion of the HDU.

`update`()

Update header keywords to reflect recent changes of columns.

`verify`(*option='warn'*)

Verify all values in the instance.

Parameters: **option** : *str*

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. May also be any combination of `"fix"` or `"silentfix"` with `"+ignore"`, `"+warn"`, or `"+exception"` (e.g. `"fix+warn"`). See Verification Options for more info.

`verify_checksum` ()

Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU CHECKSUM.

Returns: **valid** : *int*

- 0 - failure
- 1 - success
- 2 - no `CHECKSUM` keyword present

`verify_datasum` ()

Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

Returns: **valid** : *int*

- 0 - failure
- 1 - success
- 2 - no `DATASUM` keyword present

`writeto` (*name*, *output_verify='exception'*, *overwrite=False*, *checksum=False*)

Works similarly to the normal writeto(), but prepends a default **`PrimaryHDU`** are required by extension HDUs (which cannot stand on their own).

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

Section

class `astropy.io.fits.`**`Section`** (*hdu*)

Bases: **`object`**

Image section.

Slices of this object load the corresponding section of an image array from the underlying FITS file on disk, and applies any BSCALE/BZERO factors.

Section slices cannot be assigned to, and modifications to a section are not saved back to the underlying file.

See the Data Sections section of the Astropy documentation for more details.

Differs

Facilities for diffing two FITS files. Includes objects for diffing entire FITS files, individual HDUs, FITS headers, or just FITS data.

Used to implement the fitsdiff program.

FITSDiff

class `astropy.io.fits.` **`FITSDiff`**(*a*, *b*, *ignore_hdus=[]*, *ignore_keywords=[]*, *ignore_comments=[]*, *ignore_fields=[]*, *numdiffs=10*, *rtol=0.0*, *atol=0.0*, *ignore_blanks=True*, *ignore_blank_cards=True*)

Bases: **`astropy.io.fits.diff._BaseDiff`**

Diff two FITS files by filename, or two **`HDUList`** objects.

`FITSDiff` objects have the following diff attributes:

- `diff_hdu_count`: If the FITS files being compared have different numbers of HDUs, this contains a 2-tuple of the number of HDUs in each file.
- `diff_hdus`: If any HDUs with the same index are different, this contains a list of 2-tuples of the HDU index and the **`HDUDiff`** object representing the differences between the two HDUs.

Parameters: **a** : *str or* ***HDUList***

The filename of a FITS file on disk, or an **HDUList** object.

b : *str or* ***HDUList***

The filename of a FITS file on disk, or an **HDUList** object to compare to the first file.

ignore_hdus : *sequence, optional*

HDU names to ignore when comparing two FITS files or HDU lists; the presence of these HDUs and their contents are ignored. Wildcard strings may also be included in the list.

ignore_keywords : *sequence, optional*

Header keywords to ignore when comparing two headers; the presence of these keywords and their values are ignored. Wildcard strings may also be included in the list.

ignore_comments : *sequence, optional*

A list of header keywords whose comments should be ignored in the comparison. May contain wildcard strings as with ignore_keywords.

ignore_fields : *sequence, optional*

The (case-insensitive) names of any table columns to ignore if any table data is to be compared.

numdiffs : *int, optional*

The number of pixel/table values to output when reporting HDU data differences. Though the count of differences is the same either way, this allows controlling the number of different values that are kept in memory or output. If a negative value is given, then numdiffs is treated as unlimited (default: 10).

rtol : *float, optional*

The relative difference to allow when comparing two float values either in header values, image arrays, or table columns (default: 0.0). Values which satisfy the expression

\[\left| a - b \right| > \text{atol} + \text{rtol} \cdot \left| b \right|\]

are considered to be different. The underlying function used for comparison is **numpy.allclose**.

New in version 2.0.

atol : *float, optional*

The allowed absolute difference. See also `rtol` parameter.

New in version 2.0.

ignore_blanks : *bool, optional*

classmethod **fromdiff**(*other*, *a*, *b*)

Returns a new Diff object of a specific subclass from an existing diff object, passing on the values for any arguments they share in common (such as ignore_keywords).

For example:

```
>>> from astropy.io import fits
>>> hdul1, hdul2 = fits.HDUList(), fits.HDUList()
>>> headera, headerb = fits.Header(), fits.Header()
>>> fd = fits.FITSDiff(hdul1, hdul2, ignore_keywords=['*'])
>>> hd = fits.HeaderDiff.fromdiff(fd, headera, headerb)
>>> list(hd.ignore_keywords)
['*']
```

property **identical**

`True` if all the `.diff_*` attributes on this diff instance are empty, implying that no differences were found.

Any subclass of `_BaseDiff` must have at least one `.diff_*` attribute, which contains a non-empty value if and only if some difference was found between the two objects being compared.

report(*fileobj=None*, *indent=0*, *overwrite=False*)

Generates a text report on the differences (if any) between two objects, and either returns it as a string or writes it to a file-like object.

Parameters: **fileobj** : *file-like object, string, or None, optional*

If `None`, this method returns the report as a string. Otherwise it returns `None` and writes the report to the given file-like object (which must have a `.write()` method at a minimum), or to a new file at the path specified.

indent : *int*

The number of 4 space tabs to indent the report.

overwrite : *bool, optional*

If `True`, overwrite the output file if it exists. Raises an `OSError` if `False` and the output file exists. Default is `False`.

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

Returns: **report** : *str or None*

HDUDiff

class `astropy.io.fits.` **`HDUDiff`**(*a*, *b*, *ignore_keywords=[]*, *ignore_comments=[]*, *ignore_fields=[]*, *numdiffs=10*, *rtol=0.0*, *atol=0.0*, *ignore_blanks=True*, *ignore_blank_cards=True*)

Bases: **`astropy.io.fits.diff._BaseDiff`**

Diff two HDU objects, including their headers and their data (but only if both HDUs contain the same type of data (image, table, or unknown).

`HDUDiff` objects have the following diff attributes:

- `diff_extnames`: If the two HDUs have different EXTNAME values, this contains a 2-tuple of the different extension names.
- `diff_extvers`: If the two HDUS have different EXTVER values, this contains a 2-tuple of the different extension versions.
- `diff_extlevels`: If the two HDUs have different EXTLEVEL values, this contains a 2-tuple of the different extension levels.
- `diff_extension_types`: If the two HDUs have different XTENSION values, this contains a 2-tuple of the different extension types.
- `diff_headers`: Contains a `HeaderDiff` object for the headers of the two HDUs. This will always contain an object–it may be determined whether the headers are different through `diff_headers.identical`.
- `diff_data`: Contains either a `ImageDataDiff`, `TableDataDiff`, or

`RawDataDiff` as appropriate for the data in the HDUs, and only if the two HDUs have non-empty data of the same type (`RawDataDiff` is used for HDUs containing non-empty data of an indeterminate type).

Parameters:

a : *HDUList*

An **HDUList** object.

b : *str or* *HDUList*

An **HDUList** object to compare to the first **HDUList** object.

ignore_keywords : *sequence, optional*

Header keywords to ignore when comparing two headers; the presence of these keywords and their values are ignored. Wildcard strings may also be included in the list.

ignore_comments : *sequence, optional*

A list of header keywords whose comments should be ignored in the comparison. May contain wildcard strings as with ignore_keywords.

ignore_fields : *sequence, optional*

The (case-insensitive) names of any table columns to ignore if any table data is to be compared.

numdiffs : *int, optional*

The number of pixel/table values to output when reporting HDU data differences. Though the count of differences is the same either way, this allows controlling the number of different values that are kept in memory or output. If a negative value is given, then numdiffs is treated as unlimited (default: 10).

rtol : *float, optional*

The relative difference to allow when comparing two float values either in header values, image arrays, or table columns (default: 0.0). Values which satisfy the expression
\[\left| a - b \right| > \text{atol} + \text{rtol} \cdot \left| b \right|\]
are considered to be different. The underlying function used for comparison is **numpy.allclose**.
New in version 2.0.

atol : *float, optional*

The allowed absolute difference. See also `rtol` parameter.
New in version 2.0.

ignore_blanks : *bool, optional*

Ignore extra whitespace at the end of string values either in headers or data. Extra leading whitespace is not ignored (default: True).

ignore_blank_cards : *bool, optional*

Ignore all cards that are blank, i.e. they only contain whitespace *classmethod* **`fromdiff`**(*other*, *a*, *b*)

Returns a new Diff object of a specific subclass from an existing diff object, passing on the values for any arguments they share in common (such as ignore_keywords).

For example:

```
>>> from astropy.io import fits
>>> hdul1, hdul2 = fits.HDUList(), fits.HDUList()
>>> headera, headerb = fits.Header(), fits.Header()
>>> fd = fits.FITSDiff(hdul1, hdul2, ignore_keywords=['*'])
>>> hd = fits.HeaderDiff.fromdiff(fd, headera, headerb)
>>> list(hd.ignore_keywords)
['*']
```

property **`identical`**

`True` if all the `.diff_*` attributes on this diff instance are empty, implying that no differences were found.

Any subclass of `_BaseDiff` must have at least one `.diff_*` attribute, which contains a non-empty value if and only if some difference was found between the two objects being compared.

`report`(*fileobj=None*, *indent=0*, *overwrite=False*)

Generates a text report on the differences (if any) between two objects, and either returns it as a string or writes it to a file-like object.

Parameters: **fileobj** : *file-like object, string, or None, optional*

If **`None`**, this method returns the report as a string. Otherwise it returns **`None`** and writes the report to the given file-like object (which must have a `.write()` method at a minimum), or to a new file at the path specified.

indent : *int*

The number of 4 space tabs to indent the report.

overwrite : *bool, optional*

If `True`, overwrite the output file if it exists. Raises an `OSError` if `False` and the output file exists. Default is `False`.
Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

Returns: **report** : *str or None*

HeaderDiff

class `astropy.io.fits.` **`HeaderDiff`** (*a*, *b*, *ignore_keywords=[]*, *ignore_comments=[]*, *rtol=0.0*, *atol=0.0*, *ignore_blanks=True*, *ignore_blank_cards=True*)

Bases: **`astropy.io.fits.diff._BaseDiff`**

Diff two **`Header`** objects.

`HeaderDiff` objects have the following diff attributes:

- `diff_keyword_count` : If the two headers contain a different number of keywords, this contains a 2-tuple of the keyword count for each header.
- `diff_keywords` : If either header contains one or more keywords that don't appear at all in the other header, this contains a 2-tuple consisting of a list of the keywords only appearing in header a, and a list of the keywords only appearing in header b.
- `diff_duplicate_keywords` : If a keyword appears in both headers at least once, but contains a different number of duplicates (for example, a different number of HISTORY cards in each header), an item is added to this dict with the keyword as the key, and a 2-tuple of the different counts of that keyword as the value. For example:

  ```
  {'HISTORY': (20, 19)}
  ```

  means that header a contains 20 HISTORY cards, while header b contains only 19 HISTORY cards.
- `diff_keyword_values` : If any of the common keyword between the two headers have different values, they appear in this dict. It has a structure similar to `diff_duplicate_keywords`, with the keyword as the key, and a 2-tuple of the different values as the value. For example:

  ```
  {'NAXIS': (2, 3)}
  ```

  means that the NAXIS keyword has a value of 2 in header a, and a value of 3 in header b. This excludes any keywords matched by the `ignore_keywords` list.
- `diff_keyword_comments` : Like `diff_keyword_values`, but contains differences between keyword comments.

`HeaderDiff` objects also have a `common_keywords` attribute that lists all keywords that appear in both headers.

Parameters: **a** : *HDUList*

An **HDUList** object.

b : *HDUList*

An **HDUList** object to compare to the first **HDUList** object.

ignore_keywords : *sequence, optional*

Header keywords to ignore when comparing two headers; the presence of these keywords and their values are ignored. Wildcard strings may also be included in the list.

ignore_comments : *sequence, optional*

A list of header keywords whose comments should be ignored in the comparison. May contain wildcard strings as with ignore_keywords.

numdiffs : *int, optional*

The number of pixel/table values to output when reporting HDU data differences. Though the count of differences is the same either way, this allows controlling the number of different values that are kept in memory or output. If a negative value is given, then numdiffs is treated as unlimited (default: 10).

rtol : *float, optional*

The relative difference to allow when comparing two float values either in header values, image arrays, or table columns (default: 0.0). Values which satisfy the expression

\[\left| a - b \right| > \text{atol} + \text{rtol} \cdot \left| b \right|\]

are considered to be different. The underlying function used for comparison is **numpy.allclose**.

New in version 2.0.

atol : *float, optional*

The allowed absolute difference. See also `rtol` parameter.

New in version 2.0.

ignore_blanks : *bool, optional*

Ignore extra whitespace at the end of string values either in headers or data. Extra leading whitespace is not ignored (default: True).

ignore_blank_cards : *bool, optional*

Ignore all cards that are blank, i.e. they only contain whitespace (default: True).

classmethod **fromdiff**(*other*, *a*, *b*)

Returns a new Diff object of a specific subclass from an existing diff object, passing on the values for any arguments they share in common (such as ignore_keywords).

For example:

```
>>> from astropy.io import fits
>>> hdul1, hdul2 = fits.HDUList(), fits.HDUList()
>>> headera, headerb = fits.Header(), fits.Header()
>>> fd = fits.FITSDiff(hdul1, hdul2, ignore_keywords=['*'])
>>> hd = fits.HeaderDiff.fromdiff(fd, headera, headerb)
>>> list(hd.ignore_keywords)
['*']
```

property **`identical`**

`True` if all the `.diff_*` attributes on this diff instance are empty, implying that no differences were found.

Any subclass of `_BaseDiff` must have at least one `.diff_*` attribute, which contains a non-empty value if and only if some difference was found between the two objects being compared.

`report`(*fileobj=None*, *indent=0*, *overwrite=False*)

Generates a text report on the differences (if any) between two objects, and either returns it as a string or writes it to a file-like object.

Parameters: **fileobj** : *file-like object, string, or None, optional*

If **`None`**, this method returns the report as a string. Otherwise it returns **`None`** and writes the report to the given file-like object (which must have a `.write()` method at a minimum), or to a new file at the path specified.

indent : *int*

The number of 4 space tabs to indent the report.

overwrite : *bool, optional*

If `True`, overwrite the output file if it exists. Raises an `OSError` if `False` and the output file exists. Default is `False`.
Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

Returns: **report** : *str or None*

ImageDataDiff

class `astropy.io.fits.` **`ImageDataDiff`** (*a*, *b*, *numdiffs=10*, *rtol=0.0*, *atol=0.0*)

Bases: **`astropy.io.fits.diff._BaseDiff`**

Diff two image data arrays (really any array from a PRIMARY HDU or an IMAGE extension HDU, though the data unit is assumed to be “pixels”).

`ImageDataDiff` objects have the following diff attributes:

- `diff_dimensions` : If the two arrays contain either a different number of dimensions or different sizes in any dimension, this contains a 2-tuple of the shapes of each array. Currently no further comparison is performed on images that don’t have the exact same dimensions.
- `diff_pixels` : If the two images contain any different pixels, this contains a list of 2-tuples of the array index where the difference was found, and another 2-tuple containing the different values. For example, if the pixel at (0, 0) contains different values this would look like:

  ```
  [(0, 0), (1.1, 2.2)]
  ```

  where 1.1 and 2.2 are the values of that pixel in each array. This array only contains up to `self.numdiffs` differences, for storage efficiency.
- `diff_total` : The total number of different pixels found between the arrays. Although `diff_pixels` does not necessarily contain all the different pixel values, this can be used to get a count of the total number of differences found.
- `diff_ratio` : Contains the ratio of `diff_total` to the total number of pixels in the arrays.

Parameters: **a** : *HDUList*

An **HDUList** object.

b : *HDUList*

An **HDUList** object to compare to the first **HDUList** object.

numdiffs : *int, optional*

The number of pixel/table values to output when reporting HDU data differences. Though the count of differences is the same either way, this allows controlling the number of different values that are kept in memory or output. If a negative value is given, then numdiffs is treated as unlimited (default: 10).

rtol : *float, optional*

The relative difference to allow when comparing two float values either in header values, image arrays, or table columns (default: 0.0). Values which satisfy the expression

\[\left| a - b \right| > \text{atol} + \text{rtol} \cdot \left| b \right|\]

are considered to be different. The underlying function used for comparison is **numpy.allclose**.

New in version 2.0.

atol : *float, optional*

The allowed absolute difference. See also `rtol` parameter.

New in version 2.0.

classmethod **fromdiff**(*other*, *a*, *b*)

Returns a new Diff object of a specific subclass from an existing diff object, passing on the values for any arguments they share in common (such as ignore_keywords).

For example:

```
>>> from astropy.io import fits
>>> hdul1, hdul2 = fits.HDUList(), fits.HDUList()
>>> headera, headerb = fits.Header(), fits.Header()
>>> fd = fits.FITSDiff(hdul1, hdul2, ignore_keywords=['*'])
>>> hd = fits.HeaderDiff.fromdiff(fd, headera, headerb)
>>> list(hd.ignore_keywords)
['*']
```

property **identical**

True if all the `.diff_*` attributes on this diff instance are empty, implying that no differences were found.

Any subclass of `_BaseDiff` must have at least one `.diff_*` attribute.

which contains a non-empty value if and only if some difference was found between the two objects being compared.

`report`(*fileobj=None*, *indent=0*, *overwrite=False*)

Generates a text report on the differences (if any) between two objects, and either returns it as a string or writes it to a file-like object.

Parameters: **fileobj** : *file-like object, string, or None, optional*

If **None**, this method returns the report as a string. Otherwise it returns **None** and writes the report to the given file-like object (which must have a `.write()` method at a minimum), or to a new file at the path specified.

indent : *int*

The number of 4 space tabs to indent the report.

overwrite : *bool, optional*

If `True`, overwrite the output file if it exists. Raises an `OSError` if `False` and the output file exists. Default is `False`.

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

Returns: **report** : *str or None*

RawDataDiff

class `astropy.io.fits.` **`RawDataDiff`**(*a*, *b*, *numdiffs=10*)

Bases: **`astropy.io.fits.ImageDataDiff`**

`RawDataDiff` is just a special case of **`ImageDataDiff`** where the images are one-dimensional, and the data is treated as a 1-dimensional array of bytes instead of pixel values. This is used to compare the data of two non-standard extension HDUs that were not recognized as containing image or table data.

`ImageDataDiff` objects have the following diff attributes:

- `diff_dimensions`: Same as the `diff_dimensions` attribute of **`ImageDataDiff`** objects. Though the “dimension” of each array is just an integer representing the number of bytes in the data.
- `diff_bytes`: Like the `diff_pixels` attribute of **`ImageDataDiff`** objects, but renamed to reflect the minor semantic difference that these are raw bytes and not pixel values. Also the indices are integers instead of

tuples.

- `diff_total` and `diff_ratio`: Same as **`ImageDataDiff`**.

Parameters: **a** : *`HDUList`*

An **`HDUList`** object.

b : *`HDUList`*

An **`HDUList`** object to compare to the first **`HDUList`** object.

numdiffs : *int, optional*

The number of pixel/table values to output when reporting HDU data differences. Though the count of differences is the same either way, this allows controlling the number of different values that are kept in memory or output. If a negative value is given, then numdiffs is treated as unlimited (default: 10).

classmethod **`fromdiff`**(*other*, *a*, *b*)

Returns a new Diff object of a specific subclass from an existing diff object, passing on the values for any arguments they share in common (such as ignore_keywords).

For example:

```
>>> from astropy.io import fits
>>> hdul1, hdul2 = fits.HDUList(), fits.HDUList()
>>> headera, headerb = fits.Header(), fits.Header()
>>> fd = fits.FITSDiff(hdul1, hdul2, ignore_keywords=['*'])
>>> hd = fits.HeaderDiff.fromdiff(fd, headera, headerb)
>>> list(hd.ignore_keywords)
['*']
```

property **`identical`**

`True` if all the `.diff_*` attributes on this diff instance are empty, implying that no differences were found.

Any subclass of `_BaseDiff` must have at least one `.diff_*` attribute, which contains a non-empty value if and only if some difference was found between the two objects being compared.

`report`(*fileobj=None*, *indent=0*, *overwrite=False*)

Generates a text report on the differences (if any) between two objects, and either returns it as a string or writes it to a file-like object.

Parameters: **fileobj** : *file-like object, string, or None, optional*

If **None**, this method returns the report as a string. Otherwise it returns **None** and writes the report to the given file-like object (which must have a `.write()` method at a minimum), or to a new file at the path specified.

indent : *int*

The number of 4 space tabs to indent the report.

overwrite : *bool, optional*

If `True`, overwrite the output file if it exists. Raises an `OSError` if `False` and the output file exists. Default is `False`.

Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

Returns: **report** : *str or None*

TableDataDiff

class `astropy.io.fits.` **`TableDataDiff`**(*a*, *b*, *ignore_fields=[]*, *numdiffs=10*, *rtol=0.0*, *atol=0.0*)

Bases: **`astropy.io.fits.diff._BaseDiff`**

Diff two table data arrays. It doesn't matter whether the data originally came from a binary or ASCII table–the data should be passed in as a recarray.

`TableDataDiff` objects have the following diff attributes:

- `diff_column_count`: If the tables being compared have different numbers of columns, this contains a 2-tuple of the column count in each table. Even if the tables have different column counts, an attempt is still made to compare any columns they have in common.
- `diff_columns`: If either table contains columns unique to that table, either in name or format, this contains a 2-tuple of lists. The first element is a list of columns (these are full **`Column`** objects) that appear only in table a. The second element is a list of tables that appear only in table b. This only lists columns with different column definitions, and has nothing to do with the data in those columns.
- `diff_column_names`: This is like `diff_columns`, but lists only the names of columns unique to either table, rather than the full **`Column`**

objects.

- `diff_column_attributes` : Lists columns that are in both tables but have different secondary attributes, such as TUNIT or TDISP. The format is a list of 2-tuples: The first a tuple of the column name and the attribute, the second a tuple of the different values.
- `diff_values` : **`TableDataDiff`** compares the data in each table on a column-by-column basis. If any different data is found, it is added to this list. The format of this list is similar to the `diff_pixels` attribute on **`ImageDataDiff`** objects, though the "index" consists of a (column_name, row) tuple. For example:

  ```
  [('TARGET', 0), ('NGC1001', 'NGC1002')]
  ```

  shows that the tables contain different values in the 0-th row of the 'TARGET' column.
- `diff_total` and `diff_ratio` : Same as **`ImageDataDiff`**.

`TableDataDiff` objects also have a `common_columns` attribute that lists the **`Column`** objects for columns that are identical in both tables, and a `common_column_names` attribute which contains a set of the names of those columns.

Parameters: **a** : *HDUList*

An **HDUList** object.

b : *HDUList*

An **HDUList** object to compare to the first **HDUList** object.

ignore_fields : *sequence, optional*

The (case-insensitive) names of any table columns to ignore if any table data is to be compared.

numdiffs : *int, optional*

The number of pixel/table values to output when reporting HDU data differences. Though the count of differences is the same either way, this allows controlling the number of different values that are kept in memory or output. If a negative value is given, then numdiffs is treated as unlimited (default: 10).

rtol : *float, optional*

The relative difference to allow when comparing two float values either in header values, image arrays, or table columns (default: 0.0). Values which satisfy the expression
\[\left| a - b \right| > \text{atol} + \text{rtol} \cdot \left| b \right|\]
are considered to be different. The underlying function used for comparison is **numpy.allclose**.
New in version 2.0.

atol : *float, optional*

The allowed absolute difference. See also `rtol` parameter.
New in version 2.0.

classmethod **fromdiff**(*other*, *a*, *b*)

Returns a new Diff object of a specific subclass from an existing diff object, passing on the values for any arguments they share in common (such as ignore_keywords).

For example:

```
>>> from astropy.io import fits
>>> hdul1, hdul2 = fits.HDUList(), fits.HDUList()
>>> headera, headerb = fits.Header(), fits.Header()
>>> fd = fits.FITSDiff(hdul1, hdul2, ignore_keywords=['*'])
>>> hd = fits.HeaderDiff.fromdiff(fd, headera, headerb)
>>> list(hd.ignore_keywords)
['*']
```

property **identical**

True if all the `.diff_*` attributes on this diff instance are empty, implying that no differences were found.

Any subclass of `_BaseDiff` must have at least one `.diff_*` attribute, which contains a non-empty value if and only if some difference was found between the two objects being compared.

`report`(*fileobj=None*, *indent=0*, *overwrite=False*)

Generates a text report on the differences (if any) between two objects, and either returns it as a string or writes it to a file-like object.

Parameters: **fileobj** : *file-like object, string, or None, optional*

If **None**, this method returns the report as a string. Otherwise it returns **None** and writes the report to the given file-like object (which must have a `.write()` method at a minimum), or to a new file at the path specified.

indent : *int*

The number of 4 space tabs to indent the report.

overwrite : *bool, optional*

If `True`, overwrite the output file if it exists. Raises an `OSError` if `False` and the output file exists. Default is `False`.
Changed in version 1.3: `overwrite` replaces the deprecated `clobber` argument.

Returns: **report** : *str or None*

Verification Options

There are five options for the `output_verify` argument of the following methods of **HDUList**: **close()**, **writeto()**, and **flush()**, or the `_BaseHDU.writeto` method on any HDU object. In these cases, the verification option is passed to a `verify` call within these methods.

`'exception'`

This option will raise an exception if any FITS standard is violated. This is the default option for output (i.e., when **writeto()**, **close()**, or **flush()** is called). If a user wants to overwrite this default on output, the other options listed below can be used.

`'ignore'`

This option will ignore any FITS standard violation. On output, it will write the HDU List content to the output FITS file, whether or not it is conforming to FITS standard.

The `ignore` option is useful in these situations, for example:

1. An input FITS file with non-standard is read and the user wants to copy or write out after some modification to an output file. The non-standard will be preserved in such output file.
2. A user wants to create a non-standard FITS file on purpose, possibly for testing purpose.

No warning message will be printed out. This is like a silent warn (see below) option.

`'fix'`

This option will try to fix any FITS standard violations. It is not always possible to fix such violations. In general, there are two kinds of FITS standard violations: fixable and not fixable. For example, if a keyword has a floating number with an exponential notation in lower case 'e' (e.g., 1.23e11) instead of the upper case 'E' as required by the FITS standard, it is a fixable violation. On the other hand, a keyword name like `P.I.` is not fixable, since it will not know what to use to replace the disallowed periods. If a violation is fixable, this option will print out a message noting it is fixed. If it is not fixable, it will throw an exception.

The principle behind the fixing is do no harm. For example, it is plausible to 'fix' a **`Card`** with a keyword name like `P.I.` by deleting it, but `astropy` will not take such action to hurt the integrity of the data.

Not all fixes may be the "correct" fix, but at least `astropy` will try to make the fix in such a way that it will not throw off other FITS readers.

`'silentfix'`

Same as fix, but will not print out informative messages. This may be useful in a large script where the the user does not want excessive harmless messages. If the violation is not fixable, it will still throw an exception.

`'warn'`

This option is the same as the ignore option but will send warning messages. It will not try to fix any FITS standard violations whether fixable or not.

Footnotes

[1] For legacy code only that already depends on PyFITS, it's acceptable to continue using "from astropy.io import fits as pyfits".

ASCII Tables (`astropy.io.ascii`)

Introduction

`astropy.io.ascii` provides methods for reading and writing a wide range of ASCII data table formats via built-in Extension Reader Classes. The emphasis is on flexibility and convenience of use, although readers can optionally use a less flexible C-based engine for reading and writing for improved performance.

The following shows a few of the ASCII formats that are available, while the section on Supported formats contains the full list.

- **`Basic`**: basic table with customizable delimiters and header configurations
- **`Cds`**: CDS format table (also Vizier and ApJ machine readable tables)
- **`Daophot`**: table from the IRAF DAOphot package
- **`Ecsv`**: ECSV Format for lossless round-trip of data tables
- **`FixedWidth`**: table with fixed-width columns (see also Fixed-Width Gallery)
- **`Ipac`**: IPAC format table
- **`HTML`**: HTML format table contained in a <table> tag
- **`Latex`**: LaTeX table with datavalue in the `tabular` environment
- **`Rdb`**: tab-separated values with an extra line after the column definition line
- **`SExtractor`**: SExtractor format table

The strength of `astropy.io.ascii` is the support for astronomy-specific formats (often with metadata) and specialized data types such as SkyCoord, Time, and Quantity. For reading or writing large data tables in a generic format such as CSV, using the Table - Pandas interface is a recommended option to consider.

Note

It is also possible (and encouraged) to use the functionality from `astropy.io.ascii` through a higher level interface in the Data Tables package. See Unified File Read/Write Interface for more details.

Getting Started

Reading Tables

The majority of commonly encountered ASCII tables can be read with the **read()** function. Assume you have a file named `sources.dat` with the following contents:

```
obsid redshift  X      Y      object
3102  0.32      4167  4085   Q1250+568-A
877   0.22      4378  3892   "Source 82"
```

This table can be read with the following:

```
>>> from astropy.io import ascii
>>> data = ascii.read("sources.dat")
>>> print(data)
obsid redshift  X    Y      object
----- -------- ---- ---- -----------
 3102     0.32 4167 4085 Q1250+568-A
  877     0.22 4378 3892   Source 82
```

The first argument to the **read()** function can be the name of a file, a string representation of a table, or a list of table lines. The return value (`data` in this case) is a Table object.

By default, **read()** will try to guess the table format by trying all of the supported formats.

Warning

Guessing the file format is often slow for large files because the reader tries parsing the file with every allowed format until one succeeds. For large files it is recommended to disable guessing with `guess=False`.

If guessing the format does not work, as in the case for unusually formatted tables, you may need to give **astropy.io.ascii** additional hints about the format.

Examples

For unusually formatted tables, give additional hints about the format:

```
>>> lines = ['objID                   & osrcid            & xsrcid
',
...          '----------------------- & ----------------- &
-------------',
...          '              277955213 & S000.7044P00.7513 &
XS04861B6_005',
...          '              889974380 & S002.9051P14.7003 &
XS03957B7_004']
>>> data = ascii.read(lines, data_start=2, delimiter='&')
>>> print(data)
  objID         osrcid          xsrcid
--------- ----------------- -------------
277955213 S000.7044P00.7513 XS04861B6_005
889974380 S002.9051P14.7003 XS03957B7_004
```

If the format of a file is known (e.g., it is a fixed-width table or an IPAC table), then it is more efficient and reliable to provide a value for the `format` argument from one of the values in the supported formats. For example:

```
>>> data = ascii.read(lines, format='fixed_width_two_line',
delimiter='&')
```

For simpler formats such as CSV, **`read()`** will automatically try reading with the Cython/C parsing engine, which is significantly faster than the ordinary Python implementation (described in Fast ASCII I/O). If the fast engine fails, **`read()`** will fall back on the Python reader by default. The argument `fast_reader` can be specified to control this behavior. For example, to disable the fast engine:

```
>>> data = ascii.read(lines, format='csv', fast_reader=False)
```

For reading very large tables see the section on Reading Large Tables in Chunks or use pandas (see Note below).

Note

Reading a table which contains unicode characters is supported with the pure Python readers by specifying the `encoding` parameter. The fast C-readers do not support unicode. For large data files containing unicode, we recommend reading the file using pandas and converting to a Table via the Table - Pandas interface.

Writing Tables

The **`write()`** function provides a way to write a data table as a formatted ASCII table.

Examples

The following writes a table as a simple space-delimited file:

```
>>> import numpy as np
>>> from astropy.table import Table, Column, MaskedColumn
>>> x = np.array([1, 2, 3])
>>> y = x ** 2
>>> data = Table([x, y], names=['x', 'y'])
>>> ascii.write(data, 'values.dat', overwrite=True)
```

The `values.dat` file will then contain:

```
x y
1 1
2 4
3 9
```

Most of the input Reader formats supported by **`astropy.io.ascii`** for reading are also supported for writing. This provides a great deal of flexibility in the format for writing. The example below writes the data as a LaTeX table, using the option to send the output to `sys.stdout` instead of a file:

```
>>> import sys
>>> ascii.write(data, sys.stdout, format='latex')
\begin{table}
\begin{tabular}{cc}
x & y \\
1 & 1 \\
2 & 4 \\
3 & 9 \\
\end{tabular}
\end{table}
```

There is also a faster Cython engine for writing simple formats, which is enabled by default for these formats (see Fast ASCII I/O). To disable this engine, use the parameter `fast_writer`:

```
>>> ascii.write(data, 'values.csv', format='csv', fast_writer=False)
```

Finally, one can write data in the ECSV Format which allows for preserving table metadata such as column data types and units. In this way, a data table (including one with masked entries) can be stored and read back as ASCII with no loss of information.

```
>>> t = Table(masked=True)
>>> t['x'] = MaskedColumn([1.0, 2.0], unit='m', dtype='float32')
>>> t['x'][1] = np.ma.masked
>>> t['y'] = MaskedColumn([False, True], dtype='bool')
```

```
>>> import io
>>> fh = io.StringIO()
>>> t.write(fh, format='ascii.ecsv')
>>> table_string = fh.getvalue()
>>> print(table_string)
# %ECSV 0.9
# ---
# datatype:
# - {name: x, unit: m, datatype: float32}
# - {name: y, datatype: bool}
x y
1.0 False
"" True
```

```
>>> Table.read(table_string, format='ascii')
<Table masked=True length=2>
   x      y
   m
float32  bool
------- -----
    1.0 False
     --  True
```

Note

For most supported formats one can write a masked table and then read it back without losing information about the masked table entries. This is accomplished by using a blank string entry to indicate a masked (missing) value. See the Bad or Missing Values section for more information.

Supported Formats

A full list of the supported `format` values and corresponding format types for ASCII tables is given below. The `Write` column indicates which formats support write functionality, and the `Fast` column indicates which formats are compatible with the fast Cython/C engine for reading and writing.

Format	Write	Fast	Description
`aastex`	Yes		**AASTex**: AASTeX deluxetable used for AAS journals
`basic`	Yes	Yes	**Basic**: Basic table with custom delimiters

Format	Write	Fast	Description
`cds`			**`Cds`**: CDS format table
`commented_header`	Yes	Yes	**`CommentedHeader`**: Column names in a commented line
`csv`	Yes	Yes	**`Csv`**: Basic table with comma-separated values
`daophot`			**`Daophot`**: IRAF DAOphot format table
`ecsv`	Yes		**`Ecsv`**: Enhanced CSV format
`fixed_width`	Yes		**`FixedWidth`**: Fixed width
`fixed_width_no_header`	Yes		**`FixedWidthNoHeader`**: Fixed-width with no header
`fixed_width_two_line`	Yes		**`FixedWidthTwoLine`**: Fixed-width with second header line
`html`	Yes		**`HTML`**: HTML format table
`ipac`	Yes		**`Ipac`**: IPAC format table
`latex`	Yes		**`Latex`**: LaTeX table
`no_header`	Yes	Yes	**`NoHeader`**: Basic table with no headers
`rdb`	Yes	Yes	**`Rdb`**: Tab-separated with a type definition header line
`rst`	Yes		**`RST`**: reStructuredText simple format table
`sextractor`			**`SExtractor`**: SExtractor format table
`tab`	Yes	Yes	**`Tab`**: Basic table with tab-separated values

Attention

ECSV is recommended

For writing and reading tables to ASCII in a way that fully reproduces the table data, types and metadata (i.e., the table will “round-trip”), we highly recommend using the ECSV Format. This writes the actual data in a simple space-delimited format (the `basic` format) that any ASCII table reader can parse, but also includes metadata encoded in a comment block that allows full reconstruction of the original columns. This includes support for Mixin Columns (such as **`SkyCoord`** or **`Time`**) and Masked Columns.

Using `astropy.io.ascii`

The details of using **`astropy.io.ascii`** are provided in the following sections:

Reading tables

Reading Tables

The majority of commonly encountered ASCII tables can be read with the **read()** function:

```
>>> from astropy.io import ascii
>>> data = ascii.read(table)
```

Here `table` is the name of a file, a string representation of a table, or a list of table lines. The return value (`data` in this case) is a Table object.

By default, **read()** will try to guess the table format by trying all of the supported formats. Guessing the file format is often slow for large files because the reader tries parsing the file with every allowed format until one succeeds. For large files, it is recommended to disable guessing with `guess=False`.

If guessing does not work, as in the case for unusually formatted tables, then you may need to give **astropy.io.ascii** additional hints about the format:

```
>>> data = astropy.io.ascii.read('data/nls1_stackinfo.dbout',
data_start=2, delimiter='|')
>>> data = astropy.io.ascii.read('data/simple.txt', quotechar="'")
>>> data = astropy.io.ascii.read('data/simple4.txt',
format='no_header', delimiter='|')
>>> data = astropy.io.ascii.read('data/tab_and_space.txt',
delimiter=r'\s')
```

The **read()** function accepts a number of parameters that specify the detailed table format. Different formats can define different defaults, so the descriptions below sometimes mention “typical” default values. This refers to the **Basic** format reader and other similar character-separated formats.

Parameters for `read()`

table : *input table*

There are four ways to specify the table to be read:

- Path to a file (string)
- Single string containing all table lines separated by newlines
- File-like object with a callable read() method
- List of strings where each list element is a table line

The first two options are distinguished by the presence of a newline in the string. This assumes that valid file names will not normally contain a newline, and a valid table input will at least contain two rows. Note that a table read in `no_header` format can legitimately consist of a single row; in this case passing the string as a list with a single item will ensure that it is not interpreted as a file name.

format : *file format (default='basic')*

This specifies the top-level format of the ASCII table; for example, if it is a basic character delimited table, fixed format table, or a CDS-compatible table, etc. The value of this parameter must be one of the Supported Formats.

guess : *try to guess table format (default=None)*

If set to True, then **`read()`** will try to guess the table format by cycling through a number of possible table format permutations and attempting to read the table in each case. See the Guess table format section for further details.

delimiter : *column delimiter string*

A one-character string used to separate fields which typically defaults to the space character. Other common values might be "\s" (whitespace), "," or "|" or "\t" (tab). A value of "\s" allows any combination of the tab and space characters to delimit columns.

comment : *regular expression defining a comment line in table*

If the `comment` regular expression matches the beginning of a table line then that line will be discarded from header or data processing. For the `basic` format this defaults to "\s*#" (any whitespace followed by #).

quotechar : *one-character string to quote fields containing special characters*

This specifies the quote character and will typically be either the single or double quote character. This is can be useful for reading text fields with spaces in a space-delimited table. The default is typically the double quote.

header_start : *line index for the header line*

This includes only significant non-comment lines and counting starts at 0. If set to None this indicates that there is no header line and the column names will be auto-generated. See Specifying header and data location for more details.

data_start : *line index for the start of data counting*

This includes only significant non-comment lines and counting starts at 0. See Specifying header and data location for more details.

data_end : *line index for the end of data*

This includes only significant non-comment lines and can be negative to count from end. See Specifying header and data location for more details.

encoding: encoding to read the file (`default=None`)

When **`None`** use **`locale.getpreferredencoding`** as an encoding. This matches the default behavior of the built-in **`open`** when no `mode` argument is provided.

converters : `dict` *of data type converters*

See the Converters section for more information.

names : *list of names corresponding to each data column*

Define the complete list of names for each data column. This will override names found in the header (if it exists). If not supplied then use names from the header or auto-generated names if there is no header.

include_names : *list of names to include in output*

From the list of column names found from the header or the `names` parameter, select for output only columns within this list. If not supplied, then include all names.

exclude_names : *list of names to exclude from output*

Exclude these names from the list of output columns. This is applied *after* the `include_names` filtering. If not specified then no columns are excluded.

fill_values : *list of fill value specifiers*

Specify input table entries which should be masked in the output table because they are bad or missing. See the Bad or missing values section for more information and examples. The default is that any blank table values are treated as missing.

fill_include_names : *list of column names affected by* `fill_values`

This is a list of column names (found from the header or the `names` parameter) for all columns where values will be filled. **`None`** (the default) will apply `fill_values` to all columns.

fill_exclude_names : *list of column names not affected by* `fill_values`

This is a list of column names (found from the header or the `names` parameter) for all columns where values will be **not** be filled. This parameter takes precedence over `fill_include_names`. A value of **`None`** (default) does not exclude any columns.

Outputter : *Outputter class*

This converts the raw data tables value into the output object that gets returned by **`read()`**. The default is **`TableOutputter`**, which returns a **`Table`** object (see Data Tables).

Inputter : *Inputter class*

This is generally not specified.

data_Splitter : Splitter class to split data columns

header_Splitter : Splitter class to split header columns

fast_reader : *whether to use the C engine*

This can be `True` or `False`, and also be a `dict` with options. (see Fast ASCII I/O)

Reader : *Reader class (deprecated in favor of* `format`*)*

This specifies the top-level format of the ASCII table; for example, if it is a basic character delimited table, fixed format table, or a CDS-compatible table, etc. The value of this parameter must be a Reader class. For basic usage this means one of the built-in Extension Reader Classes.

Specifying Header and Data Location

The three parameters `header_start`, `data_start`, and `data_end` make it possible to read a table file that has extraneous non-table data included. This is a case where you need to help out **`astropy.io.ascii`** and tell it where to find the header and data.

When a file is processed into a header and data components, any blank lines (which might have whitespace characters) and commented lines (starting with the comment character, typically `#`) are stripped out *before* the header and data parsing code sees the table content.

Example

To use the parameters `header_start`, `data_start`, and `data_end` to read a table with non-table data included, take the file below. The column on the left is not part of the file but instead shows how **`astropy.io.ascii`** is viewing each line and the line count index.

```
Index    Table content
------
----------------------------------------------------------------
   -   | # This is the start of my data file
   -   |
   0   | Automatically generated by my_script.py at
2012-01-01T12:13:14
   1   | Run parameters: None
   2   | Column header line:
   -   |
   3   | x y z
   -   |
   4   | Data values section:
   -   |
   5   | 1 2 3
   6   | 4 5 6
   -   |
   7   | Run completed at 2012:01-01T12:14:01
```

In this case you would have `header_start=3`, `data_start=5`, and `data_end=7`. The convention for `data_end` follows the normal Python slicing convention where to select data rows 5 and 6 you would do `rows[5:7]`. For `data_end` you can also supply a negative index to count backward from the end, so `data_end=-1` (like `rows[5:-1]`) would work in this case.

Note

Prior to `astropy` v1.1, there was a bug in which a blank line that had one or more whitespace characters was mistakenly counted for `header_start` but was (correctly) not counted for `data_start` and `data_end`. If you have code that depends on the incorrect pre-1.1 behavior then it needs to be modified.

Bad or Missing Values

ASCII data tables can contain bad or missing values. A common case is when a table contains blank entries with no available data.

Examples

Take this example of a table with blank entries:

```
>>> weather_data = """
...   day,precip,type
...   Mon,1.5,rain
...   Tues,,
...   Wed,1.1,snow
...   """
```

By default, **`read()`** will interpret blank entries as being bad/missing and output a masked Table with those entries masked out by setting the corresponding mask value set to `True`:

```
>>> dat = ascii.read(weather_data)
>>> print(dat)
day  precip type
---- ------ ----
 Mon    1.5 rain
Tues     --   --
 Wed    1.1 snow
```

If you want to replace the masked (missing) values with particular values, set the masked column `fill_value` attribute and then get the "filled" version of the table. This looks like the following:

```
>>> dat['precip'].fill_value = -999
>>> dat['type'].fill_value = 'N/A'
>>> print(dat.filled())
day  precip type
---- ------ ----
 Mon    1.5 rain
Tues -999.0  N/A
 Wed    1.1 snow
```

ASCII tables may have other indicators of bad or missing data as well. For example, a table may contain string values that are not a valid representation of a number (e.g., `"..."`), or a table may have special values like `-999` that are chosen to indicate missing data. The **`read()`** function has a flexible system to accommodate these cases by marking specified character sequences in the input data as “missing data” during the conversion process. Whenever missing data is found the output will be a masked table.

This is done with the `fill_values` keyword argument, which can be set to a single missing-value specification `<missing_spec>` or a list of `<missing_spec>` tuples:

```
fill_values = <missing_spec> | [<missing_spec1>, <missing_spec2>,
...]
<missing_spec> = (<match_string>, '0', <optional col name 1>,
<optional col name 2>, ...)
```

When reading a table, the second element of a `<missing_spec>` should always be the string `'0'`, otherwise you may get unexpected behavior [1]. By default, the `<missing_spec>` is applied to all columns unless column name strings are supplied. An alternate way to limit the columns is via the `fill_include_names` and `fill_exclude_names` keyword arguments in **`read()`**.

In the example below we read back the weather table after filling the missing values in with typical placeholders:

```
>>> table = ['day   precip  type',
...          ' Mon     1.5  rain',
...          'Tues  -999.0   N/A',
...          ' Wed     1.1  snow']
>>> t = ascii.read(table, fill_values=[('-999.0', '0', 'precip'),
('N/A', '0', 'type')])
>>> print(t)
day  precip type
---- ------ ----
 Mon    1.5 rain
Tues     --   --
 Wed    1.1 snow
```

Note

The default in **read()** is `fill_values=('','0')`. This marks blank entries as being missing for any data type (int, float, or string). If `fill_values` is explicitly set in the call to **read()** then the default behavior of marking blank entries as missing no longer applies. For instance setting `fill_values=None` will disable this auto-masking without setting any other fill values. This can be useful for a string column where one of values happens to be `""`.

[1] The requirement to put the `'0'` there is the legacy of an old interface which is maintained for backward compatibility and also to match the format of `fill_value` for reading with the format of `fill_value` used for writing tables. On reading, the second element of the `<missing_spec>` tuple can actually be an arbitrary string value which replaces occurrences of the `<match_string>` string in the input stream prior to type conversion. This ends up being the value “behind the mask”, which should never be directly accessed. Only the value `'0'` is neutral when attempting to detect the column data type and perform type conversion. For instance if you used `'nan'` for the `<match_string>` value then integer columns would wind up as float.

Selecting columns for masking

The **read()** function provides the parameters `fill_include_names` and `fill_exclude_names` to select which columns will be used in the `fill_values` masking process described above.

The use of these parameters is not common but in some cases can considerably simplify the code required to read a table. The following gives a simple example to illustrate how `fill_include_names` and `fill_exclude_names` can be used in the most basic and typical cases:

```
>>> from astropy.io import ascii
>>> lines = ['a,b,c,d', '1.0,2.0,3.0,4.0', ',,,']
>>> ascii.read(lines)
<Table length=2>
   a       b       c       d
float64 float64 float64 float64
------- ------- ------- -------
    1.0     2.0     3.0     4.0
     --      --      --      --

>>> ascii.read(lines, fill_include_names=['a', 'c'])
<Table length=2>
   a     b      c     d
float64 str3 float64 str3
------- ---- ------- ----
    1.0  2.0     3.0  4.0
     --           --

>>> ascii.read(lines, fill_exclude_names=['a', 'c'])
<Table length=2>
 a      b     c      d
str3 float64 str3 float64
---- ------- ---- -------
 1.0     2.0  3.0     4.0
          --           --
```

Guess Table Format

If the `guess` parameter in **`read()`** is set to True, then **`read()`** will try to guess the table format by cycling through a number of possible table format permutations and attempting to read the table in each case. The first format which succeeds and will be used to read the table. To succeed, the table must be successfully parsed by the Reader and satisfy the following column requirements:

- At least two table columns.
- No column names are a float or int number.
- No column names begin or end with space, comma, tab, single quote, double quote, or a vertical bar (|).

These requirements reduce the chance for a false positive where a table is successfully parsed with the wrong format. A common situation is a table with numeric columns but no header row, and in this case **`astropy.io.ascii`** will auto-assign column names because of the restriction on column names that look like a number.

Guess Order

The order of guessing is shown by this Python code, where `Reader` is the class which actually implements reading the different file formats:

```
for Reader in (Ecsv, FixedWidthTwoLine, Rst, FastBasic, Basic,
               FastRdb, Rdb, FastTab, Tab, Cds, Daophot, SExtractor,
               Ipac, Latex, AASTex):
    read(Reader=Reader)

for Reader in (CommentedHeader, FastBasic, Basic, FastNoHeader,
NoHeader):
    for delimiter in ("|", ",", " ", "\\s"):
        for quotechar in ('"', "'"):
            read(Reader=Reader, delimiter=delimiter,
quotechar=quotechar)
```

Note that the **FixedWidth** derived-readers are not included in the default guess sequence (this causes problems), so to read such tables you must explicitly specify the format with the `format` keyword. Also notice that formats compatible with the fast reading engine attempt to use the fast engine before the ordinary reading engine.

If none of the guesses succeed in reading the table (subject to the column requirements), a final try is made using just the user-supplied parameters but without checking the column requirements. In this way, a table with only one column or column names that look like a number can still be successfully read.

The guessing process respects any values of the Reader, delimiter, and quotechar parameters as well as options for the fast reader that were supplied to the read() function. Any guesses that would conflict are skipped. For example, the call:

```
>>> data = ascii.read(table, Reader=ascii.NoHeader, quotechar="'")
```

would only try the four delimiter possibilities, skipping all the conflicting Reader and quotechar combinations. Similarly, with any setting of `fast_reader` that requires use of the fast engine, only the fast variants in the Reader list above will be tried.

Disabling

Guessing can be disabled in two ways:

```
import astropy.io.ascii
data = astropy.io.ascii.read(table)               # guessing enabled
by default
data = astropy.io.ascii.read(table, guess=False)  # disable for this
call
astropy.io.ascii.set_guess(False)                 # set default to
False globally
```

```
data = astropy.io.ascii.read(table)                    # guessing disabled
```

Debugging

In order to get more insight into the guessing process and possibly debug if something is not working as expected, use the `get_read_trace()` function. This returns a traceback of the attempted read formats for the last call to `read()`.

Comments and Metadata

Any comment lines detected during reading are inserted into the output table via the `comments` key in the table’s `.meta` dictionary.

Example

Comment lines detected during reading are inserted into the output table as such:

```
>>> table='''# TELESCOPE = 30 inch
...          # TARGET = PV Ceph
...          # BAND = V
...          MJD mag
...          55555 12.3
...          55556 12.4'''
>>> dat = ascii.read(table)
>>> print(dat.meta['comments'])
['TELESCOPE = 30 inch', 'TARGET = PV Ceph', 'BAND = V']
```

While `astropy.io.ascii` will not do any post-processing on comment lines, custom post-processing can be accomplished by rereading with the metadata line comments. Here is one example, where comments are of the form “# KEY = VALUE”:

```
>>> header = ascii.read(dat.meta['comments'], delimiter='=',
...                     format='no_header', names=['key', 'val'])
>>> print(header)
   key       val
--------- -------
TELESCOPE 30 inch
   TARGET PV Ceph
     BAND       V
```

Converters

`astropy.io.ascii` converts the raw string values from the table into numeric data types by using converter functions such as the Python `int` and `float` functions. For example, `int("5.0")` will fail while float(“5.0”) will succeed and return 5.0 as a Python float.

The default converters are:

```
default_converters = [astropy.io.ascii.convert_numpy(numpy.int),
                      astropy.io.ascii.convert_numpy(numpy.float),
                      astropy.io.ascii.convert_numpy(numpy.str)]
```

These take advantage of the **convert_numpy()** function which returns a two-element tuple `(converter_func, converter_type)` as described in the previous section. The type provided to **convert_numpy()** must be a valid NumPy type such as `numpy.int`, `numpy.uint`, `numpy.int8`, `numpy.int64`, `numpy.float`, `numpy.float64`, or `numpy.str`.

The default converters for each column can be overridden with the `converters` keyword:

```
>>> import numpy as np
>>> converters = {'col1': [ascii.convert_numpy(np.uint)],
...               'col2': [ascii.convert_numpy(np.float32)]}
>>> ascii.read('file.dat', converters=converters)
```

Fortran-Style Exponents

The fast converter available with the C input parser provides an `exponent_style` option to define a custom character instead of the standard `'e'` for exponential formats in the input file, to read, for example, Fortran-style double precision numbers like `'1.495978707D+13'`:

```
>>> ascii.read('double.dat', format='basic', guess=False,
...            fast_reader={'exponent_style': 'D'})
```

The special setting `'fortran'` is provided to allow for the auto-detection of any valid Fortran exponent character (`'E'`, `'D'`, `'Q'`), as well as of triple-digit exponents prefixed with no character at all (e.g., `'2.1127123261674622-107'`). All values and exponent characters in the input data are case-insensitive; any value other than the default `'E'` implies the automatic setting of `'use_fast_converter': True`.

Advanced Customization

Here we provide a few examples that demonstrate how to extend the base functionality to handle special cases. To go beyond these examples, the best reference is to read the code for the existing Extension Reader Classes.

Examples

For special cases, these examples demonstrate how to extend the base functionality of **astropy.io.ascii**.

Define custom readers by class inheritance

The most useful way to define a new reader class is by inheritance. This is the way all of the built-in readers are defined, so there are plenty of examples in the code.

In most cases, you will define one class to handle the header, one class that handles the data, and a reader class that ties it all together. Here is an example from the code that defines a reader that is just like the basic reader, but header and data start in different lines of the file:

```
# Note: NoHeader is already included in astropy.io.ascii for
convenience.
class NoHeaderHeader(BasicHeader):
    '''Reader for table header without a header

    Set the start of header line number to `None`, which tells the
basic
    reader there is no header line.
    '''
    start_line = None

class NoHeaderData(BasicData):
    '''Reader for table data without a header

    Data starts at first uncommented line since there is no header
line.
    '''
    start_line = 0

class NoHeader(Basic):
    """Read a table with no header line.  Columns are autonamed using
    header.auto_format which defaults to "col%d".  Otherwise this
reader
    the same as the :class:`Basic` class from which it is derived.
Example::

      # Table data
      1 2 "hello there"
      3 4 world
    """
    _format_name = 'no_header'
    _description = 'Basic table with no headers'
    header_class = NoHeaderHeader
    data_class = NoHeaderData
```

In a slightly more involved case, the implementation can also override some of the methods in the base class:

```
# Note: CommentedHeader is already included in astropy.io.ascii for
convenience.
class CommentedHeaderHeader(BasicHeader):
    """Header class for which the column definition line starts with
the
    comment character.  See the :class:`CommentedHeader` class  for
an example.
    """
    def process_lines(self, lines):
        """Return only lines that start with the comment regexp.  For
these
        lines strip out the matching characters."""
        re_comment = re.compile(self.comment)
        for line in lines:
            match = re_comment.match(line)
            if match:
                yield line[match.end():]

    def write(self, lines):
        lines.append(self.write_comment +
self.splitter.join(self.colnames))


class CommentedHeader(Basic):
    """Read a file where the column names are given in a line that
begins with
    the header comment character. ``header_start`` can be used to
specify the
    line index of column names, and it can be a negative index (for
example -1
    for the last commented line).  The default delimiter is the
<space>
    character.::

      # col1 col2 col3
      # Comment line
      1 2 3
      4 5 6
    """
    _format_name = 'commented_header'
    _description = 'Column names in a commented line'

    header_class = CommentedHeaderHeader
    data_class = NoHeaderData
```

Define a custom reader functionally

Instead of defining a new class, it is also possible to obtain an instance of a reader, and then to modify the properties of this one reader instance in a function:

```
def read_rdb_table(table):
    reader = astropy.io.ascii.Basic()
    reader.header.splitter.delimiter = '\t'
    reader.data.splitter.delimiter = '\t'
    reader.header.splitter.process_line = None
    reader.data.splitter.process_line = None
    reader.data.start_line = 2

    return reader.read(table)
```

Create a custom splitter.process_val function

```
# The default process_val() normally just strips whitespace.
# In addition have it replace empty fields with -999.
def process_val(x):
    """Custom splitter process_val function: Remove whitespace at the
beginning
    or end of value and substitute -999 for any blank entries."""
    x = x.strip()
    if x == '':
        x = '-999'
    return x

# Create an RDB reader and override the splitter.process_val function
rdb_reader = astropy.io.ascii.get_reader(Reader=astropy.io.ascii.Rdb)
rdb_reader.data.splitter.process_val = process_val
```

Reading Large Tables in Chunks

The default process for reading ASCII tables is not memory efficient and may temporarily require much more memory than the size of the file (up to a factor of 5 to 10). In cases where the temporary memory requirement exceeds available memory this can cause significant slowdown when disk cache gets used.

In this situation, there is a way to read the table in smaller chunks which are limited in size. There are two possible ways to do this:

- Read the table in chunks and aggregate the final table along the way. This uses only somewhat more memory than the final table requires.
- Use a Python generator function to return a `Table` object for each chunk of the input table. This allows for scanning through arbitrarily large tables since it never returns the final aggregate table.

The chunk reading functionality is most useful for very large tables, so this is available only for the Fast ASCII I/O readers. The following formats are supported: `tab`, `csv`, `no_header`, `rdb`, and `basic`. The `commented_header` format is not directly supported, but as a workaround one can read using the `no_header` format and explicitly supply the column names using the `names` argument.

In order to read a table in chunks you must provide the `fast_reader` keyword argument with a `dict` that includes the `chunk_size` key with the value being the approximate size (in bytes) of each chunk of the input table to read. In addition, if you provide a `chunk_generator` key which is set to `True`, then instead of returning a single table for the whole input it returns an iterator that provides a table for each chunk of the input.

Examples

To read an entire table while limiting peak memory usage:

```
# Read a large CSV table in 100 Mb chunks.

tbl = ascii.read('large_table.csv', format='csv', guess=False,
                 fast_reader={'chunk_size': 100 * 1000000})
```

To read the table in chunks with an iterator, we iterate over a CSV table and select all rows where the `Vmag` column is less than 8.0 (e.g., all stars in table brighter than 8.0 mag). We collect all of these subtables and then stack them at the end.

```
from astropy.table import vstack

# tbls is an iterator over the chunks (no actual reading done yet)
tbls = ascii.read('large_table.csv', format='csv', guess=False,
                  fast_reader={'chunk_size': 100 * 1000000,
                               'chunk_generator': True})

out_tbls = []

# At this point the file is actually read in chunks.
for tbl in tbls:
    bright = tbl['Vmag'] < 8.0
    if np.count_nonzero(bright):
        out_tbls.append(tbl[bright])

out_tbl = vstack(out_tbls)
```

Note

Performance

Specifying the `format` explicitly and using `guess=False` is a good idea for large tables. This prevents unnecessary guessing in the typical case where the format is already known.

The `chunk_size` should generally be set to the largest value that is reasonable given available system memory. There is overhead associated with processing each chunk, so the fewer chunks the better.

Writing tables

Writing Tables

`astropy.io.ascii` is able to write ASCII tables out to a file or file-like object using the same class structure and basic user interface as for reading tables.

The **`write()`** function provides a way to write a data table as a formatted ASCII table.

Examples

To write a formatted ASCII table using the **`write()`** function:

```
>>> import numpy as np
>>> from astropy.io import ascii
>>> x = np.array([1, 2, 3])
>>> y = x ** 2
>>> ascii.write([x, y], 'values.dat', names=['x', 'y'],
overwrite=True)
```

The `values.dat` file will then contain:

```
x y
1 1
2 4
3 9
```

Most of the input table Supported Formats for reading are also available for writing. This provides a great deal of flexibility in the format for writing. The example below writes the data as a LaTeX table, using the option to send the output to `sys.stdout` instead of a file:

```
>>> ascii.write(data, format='latex')
\begin{table}
\begin{tabular}{cc}
```

```
x & y \\
1 & 1 \\
2 & 4 \\
3 & 9 \\
\end{tabular}
\end{table}
```

There is also a faster Cython engine for writing simple formats, which is enabled by default for these formats (see Fast ASCII I/O). To disable this engine, use the parameter `fast_writer`:

```
>>> ascii.write(data, 'values.csv', format='csv', fast_writer=False)
```

Input Data Format

The input `table` argument to **`write()`** can be any value that is supported for initializing a **`Table`** object. This is documented in detail in the Constructing a Table section and includes creating a table with a list of columns, a dictionary of columns, or from **`numpy`** arrays (either structured or homogeneous).

Table or NumPy Structured Array

An Astropy **`Table`** object or a NumPy structured array (or record array) can serve as input to the **`write()`** function.

Example

To create a table with a `numpy` structured array or an existing table:

```
>>> from astropy.io import ascii
>>> from astropy.table import Table

>>> data = Table({'a': [1, 2, 3],
...               'b': [4.0, 5.0, 6.0]},
...              names=['a', 'b'])
>>> ascii.write(data)
a b
1 4.0
2 5.0
3 6.0

>>> data = np.array([(1, 2., 'Hello'), (2, 3., "World")],
...                 dtype=('i4,f4,a10'))
>>> ascii.write(data)
f0 f1 f2
1 2.0 Hello
2 3.0 World
```

The output of **`astropy.io.ascii.read`** is a **`Table`** or NumPy array data object that can be an input to the **write()** function.

```
>>> data = ascii.read('data/daophot.dat', format='daophot')
>>> ascii.write(data, 'space_delimited_table.dat')
```

List of `list` Objects

A list of Python `list` objects (or any iterable object) can be used as input:

```
>>> x = [1, 2, 3]
>>> y = [4, 5.2, 6.1]
>>> z = ['hello', 'world', '!!!']
>>> data = [x, y, z]

>>> ascii.write(data)
col0 col1 col2
1 4.0 hello
2 5.2 world
3 6.1 !!!
```

The `data` object does not contain information about the column names so **Table** has chosen them automatically. To specify the names, provide the `names` keyword argument. This example also shows excluding one of the columns from the output:

```
>>> ascii.write(data, names=['x', 'y', 'z'], exclude_names=['y'])
x z
1 hello
2 world
3 !!!
```

`dict` of `list` Objects

A dictionary containing iterable objects can serve as input to **write()**. Each dict key is taken as the column name while the value must be an iterable object containing the corresponding column values.

Since a Python dictionary is not ordered, the output column order will be unpredictable unless the `names` argument is provided.

Example

To write a table from a `dict` of `list` objects:

```
>>> data = {'x': [1, 2, 3],
...         'y': [4, 5.2, 6.1],
...         'z': ['hello', 'world', '!!!']}
>>> ascii.write(data, names=['x', 'y', 'z'])
```

```
x y z
1 4.0 hello
2 5.2 world
3 6.1 !!!
```

Parameters for `write()`

The `write()` function accepts a number of parameters that specify the detailed output table format. Each of the Supported Formats is handled by a corresponding Writer class that can define different defaults, so the descriptions below sometimes mention "typical" default values. This refers to the `Basic` writer and other similar Writer classes.

Some output format Writer classes (e.g., `Latex` or `AASTex`) accept additional keywords that can customize the output further. See the documentation of these classes for details.

output: output specifier

There are two ways to specify the output for the write operation:

- Name of a file (string)
- File-like object (from open(), StringIO, etc.)

table: input table

Any value that is supported for initializing a `Table` object (see Constructing a Table).

format: output format (default='basic')

This specifies the format of the ASCII table to be written, such as a basic character delimited table, fixed-format table, or a CDS-compatible table, etc. The value of this parameter must be one of the Supported Formats.

delimiter: column delimiter string

A one-character string used to separate fields which typically defaults to the space character. Other common values might be "," or "|" or "\t".

comment: string defining start of a comment line in output table

For the `Basic` Writer this defaults to "# ". Which comments are written and how depends on the format chosen. The comments are defined as a list of strings in the input table `meta['comments']` element. Comments in the metadata of the given `Table` will normally be written before the header, although `CommentedHeader` writes table comments after the commented header. To disable writing comments, set `comment=False`.

formats: dict of data type converters

For each key (column name) use the given value to convert the column data to a string. If the format value is string-like, then it is used as a Python format statement (e.g., '%0.2f' % value). If it is a callable function, then that function is called with a single argument containing the column value to be converted. Example:

```
astropy.io.ascii.write(table, sys.stdout, formats={'XCENTER':
'%12.1f',
                                             'YCENTER': lambda x:
round(x, 1)},
```

names: list of names corresponding to each data column

Define the complete list of names for each data column. This will override names determined from the data table (if available). If not supplied then use names from the data table or auto-generated names.

include_names: list of names to include in output

From the list of column names found from the data table or the `names` parameter, select for output only columns within this list. If not supplied then include all names.

exclude_names: list of names to exclude from output

Exclude these names from the list of output columns. This is applied *after* the `include_names` filtering. If not specified then no columns are excluded.

fill_values: list of fill value specifiers

This can be used to fill missing values in the table or replace values with special meaning.

See the Bad or Missing Values section for more information on the syntax. The syntax is almost the same as when reading a table. There is a special value `astropy.io.ascii.masked` that is used to say "output this string for all masked values in a masked table" (the default is to use an empty string `""`):

```
>>> import sys
>>> from astropy.table import Table, Column, MaskedColumn
>>> from astropy.io import ascii
>>> t = Table([(1, 2), (3, 4)], names=('a', 'b'), masked=True)
>>> t['a'].mask = [True, False]
>>> ascii.write(t, sys.stdout)
a b
"" 3
2 4
>>> ascii.write(t, sys.stdout, fill_values=[(ascii.masked, 'N/A')])
a b
N/A 3
2 4
```

Note that when writing a table, all values are converted to strings before any value is replaced. Because `fill_values` only replaces cells that are an exact match to the specification, you need to provide the string representation (stripped of whitespace) for each value. For example, in the following commands `-99` is formatted with two digits after the comma, so we need to replace `-99.00` and not `-99`:

```
>>> t = Table([(-99, 2), (3, 4)], names=('a', 'b'))
>>> ascii.write(t, sys.stdout, fill_values = [('-99.00', 'no
data')],
...             formats={'a': '%4.2f'})
a b
"no data" 3
2.00 4
```

Similarly, if you replace a value in a column that has a fixed length format (e.g., `'f4.2'`), then the string you want to replace must have the same number of characters. In the example above, `fill_values=[(' nan',' N/A')]` would work.

fill_include_names: list of column names, which are affected by `fill_values`

If not supplied, then `fill_values` can affect all columns.

fill_exclude_names: list of column names, which are not affected by `fill_values`

If not supplied, then `fill_values` can affect all columns.

fast_writer: whether to use the fast Cython writer

If this parameter is `None` (which it is by default), **`write()`** will attempt to use the faster writer (described in Fast ASCII I/O) if possible. Specifying `fast_writer=False` disables this behavior.

Writer : *Writer class (deprecated in favor of `format`)*

This specifies the top-level format of the ASCII table to be written, such as a basic character delimited table, fixed-format table, or a CDS-compatible table, etc. The value of this parameter must be a Writer class. For basic usage this means one of the built-in Extension Reader Classes. Note that Reader classes and Writer classes are synonymous; in other words, Reader classes can also write, but for historical reasons they are often called Reader classes.

ECSV Format

The Enhanced Character-Separated Values (ECSV) format can be used to write `astropy` **`Table`** or **`QTable`** datasets to a text-only data file and then read the table back without loss of information. The format handles the key issue of serializing column specifications and table metadata by using a YAML-encoded data structure. The actual tabular data are stored in a standard character separated values (CSV) format, giving compatibility with a wide variety of non-specialized CSV table readers.

Mixin Columns

Starting with `astropy` 2.0 it is possible to store not only standard **`Column`** objects to ECSV but also the following Mixin Columns:

- **`astropy.time.Time`**
- **`astropy.time.TimeDelta`**
- **`astropy.units.Quantity`**
- **`astropy.coordinates.Latitude`**
- **`astropy.coordinates.Longitude`**
- **`astropy.coordinates.Angle`**
- **`astropy.coordinates.Distance`**
- **`astropy.coordinates.EarthLocation`**
- **`astropy.coordinates.SkyCoord`**

In general, a mixin column may contain multiple data components as well as object attributes beyond the standard **`Column`** attributes like `format` or `description`. Storing such mixin columns is done by replacing the mixin column with column(s) representing the underlying data component(s) and then inserting metadata which informs the reader of how to reconstruct the original column. For example, a **`SkyCoord`** mixin column in `'spherical'` representation would have data attributes `ra`, `dec`, `distance`, along with object attributes like `representation_type` or `frame`.

Example

Creating a table with a **`SkyCoord`** column can be accomplished with a mixin column, which is supported by ECSV. To store a mixin column:

```
>>> from astropy.io import ascii
>>> from astropy.coordinates import SkyCoord
>>> import astropy.units as u
>>> from astropy.time import Time
>>> from astropy.table import QTable, Column

>>> sc = SkyCoord(ra=[1,2]*u.deg, dec=[3,4]*u.deg, distance=
[5,6]*u.m,
...                  frame='fk4', obstime=Time('2000:001'))
>>> sc.info.description = 'flying circus'
>>> c = Column([1,2])
>>> q = [1,2]*u.m
>>> q.info.format = '.2f'
>>> t = QTable([c, q, sc], names=['c', 'q', 'sc'])

>>> ascii.write(t, format='ecsv')
```

```
# %ECSV 0.9
# ---
# datatype:
# - {name: c, datatype: int64}
# - {name: q, unit: m, datatype: float64}
# - {name: sc.ra, unit: deg, datatype: float64}
# - {name: sc.dec, unit: deg, datatype: float64}
# - {name: sc.distance, unit: m, datatype: float64}
# meta: !!omap
# - __serialized_columns__:
#     q:
#       __class__: astropy.units.quantity.Quantity
#       value: !astropy.table.SerializedColumn {name: q}
#     sc:
#       __class__: astropy.coordinates.sky_coordinate.SkyCoord
#       __info__: {description: flying circus}
#       dec: !astropy.table.SerializedColumn
#         __class__: astropy.coordinates.angles.Latitude
#         value: !astropy.table.SerializedColumn {name: sc.dec}
#       distance: !astropy.table.SerializedColumn
#         __class__: astropy.coordinates.distances.Distance
#         value: !astropy.table.SerializedColumn {name: sc.distance}
#       equinox: !astropy.time.Time {format: byear_str, in_subfmt:
'*', jd1: 2400000.5,
#         jd2: 33281.92345905, out_subfmt: '*', precision: 3, scale:
tai}
#       frame: fk4
#       obstime: !astropy.time.Time {format: yday, in_subfmt: '*',
jd1: 2451544.5, jd2: 0.0,
#         out_subfmt: '*', precision: 3, scale: utc}
#       ra: !astropy.table.SerializedColumn
#         __class__: astropy.coordinates.angles.Longitude
#         value: !astropy.table.SerializedColumn {name: sc.ra}
#         wrap_angle: !astropy.coordinates.Angle
#           unit: !astropy.units.Unit {unit: deg}
#           value: 360.0
#       representation: spherical
# schema: astropy-2.0
c q sc.ra sc.dec sc.distance
1 1.0 1.0 3.0 5.0
2 2.0 2.0 4.0 6.0
```

The `'__class__'` keyword gives the fully-qualified class name and must be one of the specifically allowed `astropy` classes. There is no option to add user-specified allowed classes. The `'__info__'` keyword contains values for standard **`Column`** attributes like `description` or `format`, for any mixin columns that are represented by more than one serialized column.

Masked Columns

By default, the ECSV format uses an empty (zero-length) string in the output table to represent masked or missing data in **MaskedColumn** columns. In certain cases this may not be sufficient:

- String column that contains empty (zero-length) string(s) as valid data.
- Masked data values must be stored so those values can later be unmasked.

In this case, there is an available mechanism to specify that the full data and the mask itself should be written as columns in the output table as shown in the example below. For further context see the section on Table Serialization Methods.

Example

To specify that the full data and the mask itself should be written as columns in the output table:

```
>>> from astropy.table.table_helpers import simple_table
>>> t = simple_table(masked=True)
>>> t['c'][0] = ""  # Valid empty string in data
>>> t
<Table masked=True length=3>
  a      b     c
int64 float64 str1
----- ------- ----
   --     1.0
    2     2.0   --
    3      --    e
```

Now we tell ECSV writer to output separate data and mask columns for the string column `'c'`:

```
>>> t['c'].info.serialize_method['ecsv'] = 'data_mask'
```

When this is written out, notice that the output shows all of the data values for the `'c'` column (including the masked `'d'` value) and a new column `'c.masked'`. It also stores metadata that tells the ECSV reader to interpret the `'c'` and `'c.masked'` columns as components of one **MaskedColumn** object:

```
>>> ascii.write(t, format='ecsv')
# %ECSV 0.9
# ---
# datatype:
# - {name: a, datatype: int64}
# - {name: b, datatype: float64}
# - {name: c, datatype: string}
# - {name: c.mask, datatype: bool}
# meta: !!omap
# - __serialized_columns__:
#     c:
#       __class__: astropy.table.column.MaskedColumn
#       data: !astropy.table.SerializedColumn {name: c}
#       mask: !astropy.table.SerializedColumn {name: c.mask}
# schema: astropy-2.0
a b c c.mask
"" 1.0 "" False
2 2.0 d True
3 "" e False
```

When you read this back in, the empty (zero-length) string in the first row of column `'c'` will be preserved. You can also write all of the columns out as data and mask pairs using the Unified I/O interface for tables with the `serialize_method` keyword argument:

```
>>> t.write('out.ecsv', format='ascii.ecsv',
serialize_method='data_mask')
```

In this case, all data values (including those "under the mask" in the original table) will be restored exactly when you read the file back.

Fixed-Width Gallery

Fixed-Width Gallery

Fixed-width tables are those where each column has the same width for every row in the table. This is commonly used to make tables easy to read for humans or Fortran codes. It also reduces issues with quoting and special characters, for example:

```
Col1    Col2     Col3 Col4
---- --------- ---- ----
 1.2   "hello"    1    a
 2.4 's worlds    2    2
```

There are a number of common variations in the formatting of fixed-width tables which **`astropy.io.ascii`** can read and write. The most significant difference is whether there is no header line (**`FixedWidthNoHeader`**), one header line (**`FixedWidth`**), or two header lines (**`FixedWidthTwoLine`**). Next, there are variations in the delimiter character, like whether the delimiter appears on either end (“bookends”), or if there is padding around the delimiter.

Details are available in the class API documentation, but the easiest way to understand all of the options and their interactions is by example.

Reading

FixedWidth

Nice, typical, fixed-format table:

```
>>> from astropy.io import ascii
>>> table = """
... # comment (with blank line above)
... |  Col1  |  Col2   |
... |  1.2   | "hello" |
... |  2.4   |'s worlds|
... """
>>> ascii.read(table, format='fixed_width')
<Table length=2>
  Col1     Col2
float64    str9
------- ---------
    1.2   "hello"
    2.4 's worlds
```

Typical fixed-format table with col names provided:

```
>>> table = """
... # comment (with blank line above)
... |  Col1  |  Col2   |
... |  1.2   | "hello" |
... |  2.4   |'s worlds|
... """
>>> ascii.read(table, format='fixed_width', names=['name1', 'name2'])
<Table length=2>
 name1    name2
float64    str9
------- ---------
    1.2   "hello"
    2.4 's worlds
```

Weird input table with data values chopped by col extent:

```
>>> table = """
...   Col1  |  Col2 |
...   1.2       "hello"
...   2.4   sdf's worlds
... """
>>> ascii.read(table, format='fixed_width')
<Table length=2>
  Col1    Col2
float64   str7
------- -------
    1.2    "hel
    2.4 df's wo
```

Table with double delimiters:

```
>>> table = """
... || Name ||   Phone ||         TCP||
... |  John  | 555-1234 |192.168.1.10X|
... |  Mary  | 555-2134 |192.168.1.12X|
... |   Bob  | 555-4527 | 192.168.1.9X|
... """
>>> ascii.read(table, format='fixed_width')
<Table length=3>
Name  Phone       TCP
str4   str8      str12
---- -------- ------------
John 555-1234 192.168.1.10
Mary 555-2134 192.168.1.12
 Bob 555-4527  192.168.1.9
```

Table with space delimiter:

```
>>> table = """
...  Name  --Phone-    ----TCP-----
...  John  555-1234    192.168.1.10
...  Mary  555-2134    192.168.1.12
...   Bob  555-4527     192.168.1.9
... """
>>> ascii.read(table, format='fixed_width', delimiter=' ')
<Table length=3>
Name --Phone- ----TCP-----
str4   str8      str12
---- -------- ------------
John 555-1234 192.168.1.10
Mary 555-2134 192.168.1.12
 Bob 555-4527  192.168.1.9
```

Table with no header row and auto-column naming:

Use `header_start` and `data_start` keywords to indicate no header line.

```
>>> table = """
```

```
... |  John   | 555-1234 |192.168.1.10|
... |  Mary   | 555-2134 |192.168.1.12|
... |   Bob   | 555-4527 | 192.168.1.9|
... """
>>> ascii.read(table, format='fixed_width',
...            header_start=None, data_start=0)
<Table length=3>
col1   col2       col3
str4   str8      str12
---- -------- ------------
John 555-1234 192.168.1.10
Mary 555-2134 192.168.1.12
 Bob 555-4527  192.168.1.9
```

Table with no header row and with col names provided:

Second and third rows also have hanging spaces after final "|". Use header_start and data_start keywords to indicate no header line.

```
>>> table = ["|  John   | 555-1234 |192.168.1.10|",
...          "|  Mary   | 555-2134 |192.168.1.12|  ",
...          "|   Bob   | 555-4527 | 192.168.1.9|  "]
>>> ascii.read(table, format='fixed_width',
...            header_start=None, data_start=0,
...            names=('Name', 'Phone', 'TCP'))
<Table length=3>
Name  Phone       TCP
str4   str8      str12
---- -------- ------------
John 555-1234 192.168.1.10
Mary 555-2134 192.168.1.12
 Bob 555-4527  192.168.1.9
```

FixedWidthNoHeader

Table with no header row and auto-column naming. Use the ``fixed_width_no_header`` format for convenience:

```
>>> table = """
... |  John   | 555-1234 |192.168.1.10|
... |  Mary   | 555-2134 |192.168.1.12|
... |   Bob   | 555-4527 | 192.168.1.9|
... """
>>> ascii.read(table, format='fixed_width_no_header')
<Table length=3>
col1   col2       col3
str4   str8      str12
---- -------- ------------
John 555-1234 192.168.1.10
```

```
Mary 555-2134 192.168.1.12
 Bob 555-4527  192.168.1.9
```

Table with no delimiter with column start and end values specified:

This uses the col_starts and col_ends keywords. Note that the col_ends values are inclusive so a position range of zero to five will select the first six characters.

```
>>> table = """
... #    5   9     17  18       28    <== Column start / end indexes
... #    |   |       ||         |     <== Column separation positions
...   John   555- 1234 192.168.1.10
...   Mary   555- 2134 192.168.1.12
...    Bob   555- 4527  192.168.1.9
... """
>>> ascii.read(table, format='fixed_width_no_header',
...                 names=('Name', 'Phone', 'TCP'),
...                 col_starts=(0, 9, 18),
...                 col_ends=(5, 17, 28),
...                 )
<Table length=3>
Name   Phone      TCP
str4    str9     str10
---- --------- ----------
John 555- 1234 192.168.1.
Mary 555- 2134 192.168.1.
 Bob 555- 4527  192.168.1
```

Table with no delimiter with only column start or end values specified:

If only the col_starts keyword is given, it is assumed that each column ends where the next column starts, and the final column ends at the same position as the longest line of data.

Conversely, if only the col_ends keyword is given, it is assumed that the first column starts at position zero and that each successive column starts immediately after the previous one.

The two examples below read the same table and produce the same result.

```
>>> table = """
... #1       9        19                <== Column start indexes
... #|       |         |                <== Column start positions
... #<------><--------><------------->  <== Inferred column positions
...   John   555- 1234 192.168.1.10
...   Mary   555- 2134 192.168.1.123
...    Bob   555- 4527  192.168.1.9
...    Bill  555-9875  192.255.255.255
... """
>>> ascii.read(table,
...                     format='fixed_width_no_header',
...                     names=('Name', 'Phone', 'TCP'),
...                     col_starts=(1, 9, 19),
...                     )
<Table length=4>
Name   Phone         TCP
str4    str9        str15
---- --------- ---------------
John 555- 1234    192.168.1.10
Mary 555- 2134   192.168.1.123
 Bob 555- 4527     192.168.1.9
Bill  555-9875 192.255.255.255

>>> ascii.read(table,
...                     format='fixed_width_no_header',
...                     names=('Name', 'Phone', 'TCP'),
...                     col_ends=(8, 18, 32),
...                     )
<Table length=4>
Name   Phone        TCP
str4    str9       str14
---- --------- --------------
John 555- 1234   192.168.1.10
Mary 555- 2134  192.168.1.123
 Bob 555- 4527    192.168.1.9
Bill  555-9875 192.255.255.25
```

FixedWidthTwoLine

Typical fixed-format table with two header lines with some cruft:

```
>>> table = """
...   Col1    Col2
...   ----  ---------
...    1.2xx"hello"
...   2.4   's worlds
... """
>>> ascii.read(table, format='fixed_width_two_line')
<Table length=2>
  Col1     Col2
float64    str9
------- ---------
    1.2   "hello"
    2.4 's worlds
```

reStructuredText table:

```
>>> table = """
... ======= ===========
...   Col1    Col2
... ======= ===========
...   1.2   "hello"
...   2.4   's worlds
... ======= ===========
... """
>>> ascii.read(table, format='fixed_width_two_line',
...                 header_start=1, position_line=2, data_end=-1)
<Table length=2>
  Col1     Col2
float64    str9
------- ---------
    1.2   "hello"
    2.4 's worlds
```

Text table designed for humans and test having position line before the header line:

```
>>> table = """
... +------+----------+
... | Col1 |   Col2   |
... +------|----------+
... |  1.2 | "hello"  |
... |  2.4 | 's worlds|
... +------+----------+
... """
>>> ascii.read(table, format='fixed_width_two_line', delimiter='+',
...                 header_start=1, position_line=0, data_start=3,
data_end=-1)
<Table length=2>
  Col1     Col2
float64    str9
------- ---------
    1.2   "hello"
    2.4 's worlds
```

Writing

FixedWidth

Define input values ``dat`` for all write examples:

```
>>> table = """
... | Col1 |  Col2     |  Col3 | Col4 |
... | 1.2  | "hello"   |  1    | a    |
... | 2.4  | 's worlds |  2    | 2    |
... """
```

```
>>> dat = ascii.read(table, format='fixed_width')
```

Write a table as a normal fixed-width table:

```
>>> ascii.write(dat, format='fixed_width')
| Col1 |      Col2 | Col3 | Col4 |
|  1.2 |   "hello" |    1 |    a |
|  2.4 | 's worlds |    2 |    2 |
```

Write a table as a fixed-width table with no padding:

```
>>> ascii.write(dat, format='fixed_width', delimiter_pad=None)
|Col1|     Col2|Col3|Col4|
| 1.2|  "hello"|   1|   a|
| 2.4|'s worlds|   2|   2|
```

Write a table as a fixed-width table with no bookend:

```
>>> ascii.write(dat, format='fixed_width', bookend=False)
Col1 |      Col2 | Col3 | Col4
 1.2 |   "hello" |    1 |    a
 2.4 | 's worlds |    2 |    2
```

Write a table as a fixed-width table with no delimiter:

```
>>> ascii.write(dat, format='fixed_width', bookend=False,
delimiter=None)
Col1       Col2  Col3  Col4
 1.2    "hello"     1     a
 2.4  's worlds     2     2
```

Write a table as a fixed-width table with no delimiter and formatting:

```
>>> ascii.write(dat, format='fixed_width',
...                     formats={'Col1': '%-8.3f', 'Col2': '%-15s'})
|     Col1 |            Col2 | Col3 | Col4 |
| 1.200    | "hello"         |    1 |    a |
| 2.400    | 's worlds       |    2 |    2 |
```

FixedWidthNoHeader

Write a table as a normal fixed-width table:

```
>>> ascii.write(dat, format='fixed_width_no_header')
| 1.2 |   "hello" | 1 | a |
| 2.4 | 's worlds | 2 | 2 |
```

Write a table as a fixed-width table with no padding:

```
>>> ascii.write(dat, format='fixed_width_no_header',
delimiter_pad=None)
|1.2|  "hello"|1|a|
|2.4|'s worlds|2|2|
```

Write a table as a fixed-width table with no bookend:

```
>>> ascii.write(dat, format='fixed_width_no_header', bookend=False)
1.2 |   "hello" | 1 | a
2.4 | 's worlds | 2 | 2
```

Write a table as a fixed-width table with no delimiter:

```
>>> ascii.write(dat, format='fixed_width_no_header', bookend=False,
...                      delimiter=None)
1.2    "hello"  1  a
2.4  's worlds  2  2
```

FixedWidthTwoLine

Write a table as a normal fixed-width table:

```
>>> ascii.write(dat, format='fixed_width_two_line')
Col1      Col2 Col3 Col4
---- --------- ---- ----
 1.2   "hello"    1    a
 2.4 's worlds    2    2
```

Write a table as a fixed width table with space padding and '=' position_char:

```
>>> ascii.write(dat, format='fixed_width_two_line',
...                      delimiter_pad=' ', position_char='=')
Col1        Col2   Col3   Col4
====   =========   ====   ====
 1.2     "hello"      1      a
 2.4   's worlds      2      2
```

Write a table as a fixed-width table with no bookend:

```
>>> ascii.write(dat, format='fixed_width_two_line', bookend=True,
delimiter='|')
|Col1|     Col2|Col3|Col4|
|----|---------|----|----|
| 1.2|  "hello"|   1|   a|
| 2.4|'s worlds|   2|   2|
```

Fast ASCII Engine

Fast ASCII I/O

While **astropy.io.ascii** was designed with flexibility and extensibility in mind, there is also a less flexible but significantly faster Cython/C engine for reading and writing ASCII files. By default, **read()** and **write()** will attempt to use this engine when dealing with compatible formats. The following formats are currently compatible with the fast engine:

- `basic`
- `commented_header`
- `csv`
- `no_header`
- `rdb`
- `tab`

The fast engine can also be enabled through the format parameter by prefixing a compatible format with “fast” and then an underscore. In this case, or when enforcing the fast engine by either setting `fast_reader='force'` or explicitly setting any of the Parallel and Fast Conversion Options, **read()** will not fall back on an ordinary reader if fast reading fails.

Examples

To open a CSV file and write it back out:

```
>>> from astropy.table import Table
>>> t = ascii.read('file.csv', format='fast_csv')
>>> t.write('output.csv', format='ascii.fast_csv')
```

To disable the fast engine, specify `fast_reader=False` or `fast_writer=False`. For example:

```
>>> t = ascii.read('file.csv', format='csv', fast_reader=False)
>>> t.write('file.csv', format='csv', fast_writer=False)
```

Note

Guessing and Fast reading

By default **read()** will try to guess the format of the input data by successively trying different formats until one succeeds (see the section on Guess Table Format). For each supported format it will first try the fast, then the slow version of that reader. Without any additional options this means that both some pure Python readers with no fast implementation and the Python versions of some readers will be tried before getting to some of the fast readers. To bypass them entirely, a fast reader should be explicitly requested as above.

For optimum performance however, it is recommended to turn off guessing entirely (`guess=False`) or narrow down the format options as much as possible by specifying the format (e.g., `format='csv'`) and/or other options such as the delimiter.

Reading

Since the fast engine is not part of the ordinary **astropy.io.ascii** infrastructure, fast readers raise an error when passed certain parameters which are not implemented in the fast reader infrastructure. In this case **read()** will fall back on the ordinary reader, unless the fast reader has been explicitly requested (see above). These parameters are:

- Negative `header_start` (except for commented-header format)
- Negative `data_start`
- `data_start=None`
- `comment` string not of length 1
- `delimiter` string not of length 1
- `quotechar` string not of length 1
- `converters`
- `Outputter`
- `Inputter`
- `data_Splitter`
- `header_Splitter`

Parallel and Fast Conversion Options

In addition to `True` and `False`, the parameter `fast_reader` can also be a `dict` specifying any of three additional parameters, `parallel`, `use_fast_converter` and `exponent_style`.

Example

To specify additional parameters using `fast_reader`:

```
>>> ascii.read('data.txt', format='basic',
...            fast_reader={'parallel': True, 'use_fast_converter':
True})
```

These options allow for even faster table reading when enabled, but both are disabled by default because they come with some caveats.

The `parallel` parameter can be used to enable multiprocessing via the `multiprocessing` module, and can either be set to a number (the number of processes to use) or `True`, in which case the number of processes will be `multiprocessing.cpu_count()`. Note that this can cause issues within the IPython Notebook and so enabling multiprocessing in this context is discouraged.

Setting `use_fast_converter` to be `True` enables a faster but slightly imprecise conversion method for floating-point values, as described below.

The `exponent_style` parameter allows to define a different character from the default `'e'` for exponential formats in the input file. The special setting `'fortran'` enables auto-detection of any valid exponent character under Fortran notation. For details see the section on Fortran-Style Exponents.

Fast Converter

Input floating-point values should ideally be converted to the nearest possible floating-point approximation; that is, the conversion should be correct within half of the distance between the two closest representable values, or 0.5 ULP. The ordinary readers, as well as the default fast reader, are guaranteed to convert floating-point values within 0.5 ULP, but there is also a faster and less accurate conversion method accessible via `use_fast_converter`. If the input data has less than about fifteen significant figures, or if accuracy is relatively unimportant, this converter might be the best option in performance-critical scenarios.

Here is an IPython notebook analyzing the error of the fast converter, both in decimal values and in ULP. For values with a reasonably small number of significant figures, the fast converter is guaranteed to produce an optimal conversion (within 0.5 ULP). Once the number of significant figures exceeds the precision of 64-bit floating-point values, the fast converter is no longer guaranteed to be within 0.5 ULP, but about 60% of values end up within 0.5 ULP and about 90% within 1.0 ULP. Another notebook analyzing the fast converter’s behavior with extreme values (such as subnormals and values out of the range of floats) is available here.

Reading Large Tables

For reading very large tables using the fast reader, see the section on Reading

Large Tables in Chunks.

Writing

The fast engine supports the same functionality as the ordinary writing engine and is generally about two to four times faster than the ordinary engine. An IPython notebook testing the relative performance of the fast writer against the ordinary writing system and the data analysis library Pandas is available here. The speed advantage of the faster engine is greatest for integer data and least for floating-point data; the fast engine is around 3.6 times faster for a sample file including a mixture of floating-point, integer, and text data. Also note that stripping string values slows down the writing process, so specifying `strip_whitespace=False` can improve performance.

Speed Gains

The fast ASCII engine was designed based on the general parsing strategy used in the Pandas data analysis library, so its performance is generally comparable (although slightly slower by default) to the Pandas `read_csv` method. Here is an IPython notebook comparing the performance of the ordinary **`astropy.io.ascii`** reader, the fast reader, the fast reader with the fast converter enabled, NumPy's `genfromtxt`, and Pandas' `read_csv` for different kinds of table data in a basic space-delimited file.

In summary, `genfromtxt` and the ordinary **`astropy.io.ascii`** reader are very similar in terms of speed, while `read_csv` is slightly faster than the fast engine for integer and floating-point data; for pure floating-point data, enabling the fast converter yields a speedup of about 50%. Also note that Pandas uses the exact same method as the fast converter in Astropy when converting floating-point data.

The difference in performance between the fast engine and Pandas for text data depends on the extent to which data values are repeated, as Pandas is almost twice as fast as the fast engine when every value is identical and the reverse is true when values are randomized. This is because the fast engine uses fixed-size NumPy string arrays for text data, while Pandas uses variable-size object arrays and uses an underlying set to avoid copying repeated values.

Overall, the fast engine tends to be around four or five times faster than the ordinary ASCII engine. If the input data is very large (generally about 100,000 rows or greater), and particularly if the data does not contain primarily integer data or repeated string values, specifying `parallel` as `True` can yield further performance gains. Although IPython does not work well with `multiprocessing`, there is a script available for testing the performance of the fast engine in parallel, and a sample result may be viewed here. This profile uses the fast converter for both the serial and parallel Astropy readers.

Another point worth noting is that the fast engine uses memory mapping if a filename is supplied as input. If you want to avoid this for whatever reason, supply an open file object instead. However, this will generally be less efficient from both a time and a memory perspective, as the entire file input will have to be read at once.

Base Class Elements

Base Class Elements

The key elements in `astropy.io.ascii` are:

- `Column`: internal storage of column properties and data.
- `Reader`: base class to handle reading and writing tables.
- `Inputter`: gets the lines from the table input.
- `Splitter`: splits the lines into string column values.
- `Header`: initializes output columns based on the table header or user input.
- `Data`: populates column data from the table.
- `Outputter`: converts column data to the specified output format (e.g., `numpy` structured array).

Each of these elements is an inheritable class with attributes that control the corresponding functionality. In this way, the large number of tunable parameters are modularized into manageable groups. In certain places these attributes are actually functions for handling special cases.

Extension Reader Classes

Extension Reader Classes

The following classes extend the base `BaseReader` functionality to handle reading and writing different table formats. Some, such as the `Basic` Reader class are fairly general and include a number of configurable attributes. Others such as `Cds` or `Daophot` are specialized to read certain well-defined but idiosyncratic formats.

- `AASTex`: AASTeX deluxetable used for AAS journals.
- `Basic`: basic table with customizable delimiters and header configurations.
- `Cds`: CDS format table (also Vizier and ApJ machine readable tables).
- `CommentedHeader`: column names given in a line that begins with the

comment character.
- `Csv`: comma-separated values.
- `Daophot`: table from the IRAF DAOphot package.
- `FixedWidth`: table with fixed-width columns (see also Fixed-Width Gallery).
- `FixedWidthNoHeader`: table with fixed-width columns and no header.
- `FixedWidthTwoLine`: table with fixed-width columns and a two-line header.
- `HTML`: HTML format table contained in a <table> tag.
- `Ipac`: IPAC format table.
- `Latex`: LaTeX table with datavalue in the `tabular` environment.
- `NoHeader`: basic table with no header where columns are auto-named.
- `Rdb`: tab-separated values with an extra line after the column definition line.
- `RST`: reStructuredText simple format table.
- `SExtractor`: SExtractor format table.
- `Tab`: tab-separated values.

Performance Tips

By default, when trying to read a file the reader will guess the format, which involves trying to read it with many different readers. For better performance when dealing with large tables, it is recommended to specify the format and any options explicitly, and turn off guessing as well.

Example

If you are reading a simple CSV file with a one-line header with column names, the following:

```
read('example.csv', format='basic', delimiter=',', guess=False)  # doctest: +SKIP
```

can be at least an order of magnitude faster than:

```
read('example.csv')  # doctest: +SKIP
```

Reference/API

astropy.io.ascii Package

An extensible ASCII table reader and writer.

Functions

convert_numpy(numpy_type)	Return a tuple containing a function which converts a list into a numpy array and the type produced by the converter function.
get_read_trace()	Return a traceback of the attempted read formats for the last call to **read** where guessing was enabled.
get_reader([Reader, Inputter, Outputter])	Initialize a table reader allowing for common customizations.
get_writer([Writer, fast_writer])	Initialize a table writer allowing for common customizations.
read(table[, guess])	Read the input `table` and return the table.
set_guess(guess)	Set the default value of the `guess` parameter for read()
write(table[, output, format, Writer, …])	Write the input `table` to `filename`.

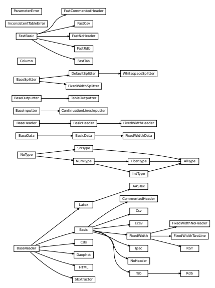
Classes

AASTex(**kwargs)	AASTeX format table.
AllType()	Subclass of all other data types.
BaseData()	Base table data reader.
BaseHeader()	Base table header reader
BaseInputter()	Get the lines from the table input and return a list of lines.
BaseOutputter()	Output table as a dict of column objects keyed on column name.
BaseReader()	Class providing methods to read and write an ASCII table using the specified header, data, inputter, and outputter instances.
BaseSplitter()	Base splitter that uses python’s split method to do the work.
Basic()	Character-delimited table with a single header line at the top.
BasicData()	Basic table Data Reader
BasicHeader()	Basic table Header Reader
Cds([readme])	CDS format table.
Column(name)	Table column.
CommentedHeader()	Character-delimited table with column names in a comment line.
ContinuationLinesInputter()	Inputter where lines ending in `continuation_char` are joined with the subsequent line. Example::.
Csv()	CSV (comma-separated-values) table.
Daophot()	DAOphot format table.
DefaultSplitter()	Default class to split strings into columns using python csv.

`Ecsv`()	ECSV (Enhanced Character Separated Values) format table.
`FastBasic`([default_kwargs])	This class is intended to handle the same format addressed by the ordinary `Basic` writer, but it acts as a wrapper for underlying C code and is therefore much faster.
`FastCommentedHeader`(**kwargs)	A faster version of the `CommentedHeader` reader, which looks for column names in a commented line.
`FastCsv`(**kwargs)	A faster version of the ordinary `Csv` writer that uses the optimized C parsing engine.
`FastNoHeader`(**kwargs)	This class uses the fast C engine to read tables with no header line.
`FastRdb`(**kwargs)	A faster version of the `Rdb` reader.
`FastTab`(**kwargs)	A faster version of the ordinary `Tab` reader that uses the optimized C parsing engine.
`FixedWidth`([col_starts, col_ends, …])	Fixed width table with single header line defining column names and positions.
`FixedWidthData`()	Base table data reader.
`FixedWidthHeader`()	Fixed width table header reader.
`FixedWidthNoHeader`([col_starts, col_ends, …])	Fixed width table which has no header line.
`FixedWidthSplitter`()	Split line based on fixed start and end positions for each `col` in `self.cols`.
`FixedWidthTwoLine`([position_line, …])	Fixed width table which has two header lines.
`FloatType`()	Describes floating-point data.
`HTML`([htmldict])	HTML format table.
`InconsistentTableError`	Indicates that an input table is inconsistent in some way.
`IntType`()	Describes integer data.
`Ipac`([definition, DBMS])	IPAC format table.
`Latex`([ignore_latex_commands, latexdict, …])	LaTeX format table.
`NoHeader`()	Character-delimited table with no header line.
`NoType`()	Superclass for `StrType` and `NumType` classes.
`NumType`()	Indicates that a column consists of numerical data.
`ParameterError`	Indicates that a reader cannot handle a passed parameter.
`RST`()	reStructuredText simple format table.
`Rdb`()	Tab-separated file with an extra line after the column definition line that specifies either numeric (N) or string (S) data.
`SExtractor`()	SExtractor format table.

StrType()	Indicates that a column consists of text data.
Tab()	Tab-separated table.
TableOutputter()	Output the table as an astropy.table.Table object.
WhitespaceSplitter()	

Class Inheritance Diagram



VOTable XML Handling (`astropy.io.votable`)

Introduction

The **`astropy.io.votable`** sub-package converts VOTable XML files to and from `numpy` record arrays.

Getting Started

Reading a VOTable File

To read in a VOTable file, pass a file path to **`parse`**:

```
from astropy.io.votable import parse
votable = parse("votable.xml")
```

`votable` is a **`VOTableFile`** object, which can be used to retrieve and manipulate the data and save it back out to disk.

VOTable files are made up of nested `RESOURCE` elements, each of which may contain one or more `TABLE` elements. The `TABLE` elements contain the arrays of data.

To get at the `TABLE` elements, you can write a loop over the resources in the `VOTABLE` file:

```
for resource in votable.resources:
    for table in resource.tables:
        # ... do something with the table ...
        pass
```

However, if the nested structure of the resources is not important, you can use **iter_tables** to return a flat list of all tables:

```
for table in votable.iter_tables():
    # ... do something with the table ...
    pass
```

Finally, if you expect only one table in the file, it might be most convenient to use **get_first_table**:

```
table = votable.get_first_table()
```

Alternatively, there is a convenience method to parse a VOTable file and return the first table all in one step:

```
from astropy.io.votable import parse_single_table
table = parse_single_table("votable.xml")
```

From a **Table** object, you can get the data itself in the `array` member variable:

```
data = table.array
```

This data is a `numpy` record array.

The columns get their names from both the `ID` and `name` attributes of the `FIELD` elements in the `VOTABLE` file.

Examples

Suppose we had a `FIELD` specified as follows:

```
<FIELD ID="Dec" name="dec_targ" datatype="char" ucd="POS_EQ_DEC_MAIN"
       unit="deg">
 <DESCRIPTION>
  representing the ICRS declination of the center of the image.
 </DESCRIPTION>
</FIELD>
```

Note

The mapping from VOTable `name` and `ID` attributes to `numpy` dtype `names` and `titles` is highly confusing.

In VOTable, `ID` is guaranteed to be unique, but is not required. `name` is not guaranteed to be unique, but is required.

In `numpy` record dtypes, `names` are required to be unique and are required. `titles` are not required, and are not required to be unique.

Therefore, VOTable's `ID` most closely maps to `numpy`'s `names`, and VOTable's `name` most closely maps to `numpy`'s `titles`. However, in some cases where a VOTable `ID` is not provided, a `numpy` `name` will be generated based on the VOTable `name`. Unfortunately, VOTable fields do not have an attribute that is both unique and required, which would be the most convenient mechanism to uniquely identify a column.

When converting from an **`astropy.io.votable.tree.Table`** object to an **`astropy.table.Table`** object, you can specify whether to give preference to `name` or `ID` attributes when naming the columns. By default, `ID` is given preference. To give `name` preference, pass the keyword argument `use_names_over_ids=True`:

```
>>> votable.get_first_table().to_table(use_names_over_ids=True)
```

This column of data can be extracted from the record array using:

```
>>> table.array['dec_targ']
array([17.15153360566, 17.15153360566, 17.15153360566, 17.1516686826,
       17.1516686826, 17.1516686826, 17.1536197136, 17.1536197136,
       17.1536197136, 17.15375479055, 17.15375479055, 17.15375479055,
       17.1553884541, 17.15539736932, 17.15539752176,
       17.25736014763,
       # ...
       17.2765703], dtype=object)
```

or equivalently:

```
>>> table.array['Dec']
array([17.15153360566, 17.15153360566, 17.15153360566, 17.1516686826,
       17.1516686826, 17.1516686826, 17.1536197136, 17.1536197136,
       17.1536197136, 17.15375479055, 17.15375479055, 17.15375479055,
       17.1553884541, 17.15539736932, 17.15539752176,
       17.25736014763,
       # ...
```

```
        17.2765703], dtype=object)
```

Building a New Table from Scratch

It is also possible to build a new table, define some field datatypes, and populate it with data.

Example

To build a new table from a VOTable file:

```
from astropy.io.votable.tree import VOTableFile, Resource, Table,
Field

# Create a new VOTable file...
votable = VOTableFile()

# ...with one resource...
resource = Resource()
votable.resources.append(resource)

# ... with one table
table = Table(votable)
resource.tables.append(table)

# Define some fields
table.fields.extend([
        Field(votable, name="filename", datatype="char",
arraysize="*"),
        Field(votable, name="matrix", datatype="double",
arraysize="2x2")])

# Now, use those field definitions to create the numpy record arrays,
with
# the given number of rows
table.create_arrays(2)

# Now table.array can be filled with data
table.array[0] = ('test1.xml', [[1, 0], [0, 1]])
table.array[1] = ('test2.xml', [[0.5, 0.3], [0.2, 0.1]])

# Now write the whole thing to a file.
# Note, we have to use the top-level votable file object
votable.to_xml("new_votable.xml")
```

Outputting a VOTable File

To save a VOTable file, call the **to_xml** method. It accepts either a string or Unicode path, or a Python file-like object:

```
votable.to_xml('output.xml')
```

There are a number of data storage formats supported by **astropy.io.votable**. The `TABLEDATA` format is XML-based and stores values as strings representing numbers. The `BINARY` format is more compact, and stores numbers in base64-encoded binary. VOTable version 1.3 adds the `BINARY2` format, which allows for masking of any data type, including integers and bit fields which cannot be masked in the older `BINARY` format. The storage format can be set on a per-table basis using the **format** attribute, or globally using the **set_all_tables_format** method:

```
votable.get_first_table().format = 'binary'
votable.set_all_tables_format('binary')
votable.to_xml('binary.xml')
```

Using astropy.io.votable

Standard Compliance

astropy.io.votable.tree.Table supports the VOTable Format Definition Version 1.1, Version 1.2, and the Version 1.3 proposed recommendation. Some flexibility is provided to support the 1.0 draft version and other nonstandard usage in the wild. To support these cases, set the keyword argument `pedantic` to `False` when parsing.

Note

Each warning and VOTABLE-specific exception emitted has a number and is documented in more detail in Warnings and Exceptions.

Output always conforms to the 1.1, 1.2, or 1.3 spec, depending on the input.

Pedantic mode

Many VOTable files in the wild do not conform to the VOTable specification. If reading one of these files causes exceptions, you may turn off pedantic mode in **astropy.io.votable** by passing `pedantic=False` to the **parse** or **parse_single_table** functions:

```
from astropy.io.votable import parse
votable = parse("votable.xml", pedantic=False)
```

Note, however, that it is good practice to report these errors to the author of the application that generated the VOTable file to bring the file into compliance with the specification.

Even with `pedantic` turned off, many warnings may still be omitted. These warnings are all of the type **`VOTableSpecWarning`** and can be turned off using the standard Python **`warnings`** module.

Missing Values

Any value in the table may be "missing". **`astropy.io.votable`** stores a `numpy` masked array in each **`Table`** instance. This behaves like an ordinary `numpy` masked array, except for variable-length fields. For those fields, the datatype of the column is "object" and another `numpy` masked array is stored there. Therefore, operations on variable-length columns will not work — this is because variable-length columns are not directly supported by `numpy` masked arrays.

Datatype Mappings

The datatype specified by a `FIELD` element is mapped to a `numpy` type according to the following table:

VOTABLE type	NumPy type
boolean	b1
bit	b1
unsignedByte	u1
char (*variable length*)	O - A `bytes()` object.
char (*fixed length*)	S
unicodeChar (*variable length*)	O - A **`str`** object
unicodeChar (*fixed length*)	U
short	i2
int	i4
long	i8
float	f4
double	f8
floatComplex	c8

VOTABLE type	NumPy type
doubleComplex	c16

If the field is a fixed-size array, the data is stored as a `numpy` fixed-size array.

If the field is a variable-size array (that is, `arraysize` contains a '*'), the cell will contain a Python list of `numpy` values. Each value may be either an array or scalar depending on the `arraysize` specifier.

Examining Field Types

To look up more information about a field in a table, you can use the **`get_field_by_id`** method, which returns the **`Field`** object with the given ID.

Example

To look up more information about a field:

```
>>> field = table.get_field_by_id('Dec')
>>> field.datatype
'char'
>>> field.unit
'deg'
```

Note

Field descriptors should not be mutated. To change the set of columns, convert the Table to an **`astropy.table.Table`**, make the changes, and then convert it back.

Data Serialization Formats

VOTable supports a number of different serialization formats.

- TABLEDATA stores the data in pure XML, where the numerical values are written as human-readable strings.
- BINARY is a binary representation of the data, stored in the XML as an opaque `base64` -encoded blob.
- BINARY2 was added in VOTable 1.3, and is identical to "BINARY", except that it explicitly records the position of missing values rather than identifying them by a special value.
- FITS stores the data in an external FITS file. This serialization is not supported by the **`astropy.io.votable`** writer, since it requires writing multiple files.

The serialization format can be selected in two ways:

1) By setting the `format` attribute of a **`astropy.io.votable.tree.Table`** object:

```
votable.get_first_table().format = "binary"
votable.to_xml("new_votable.xml")
```

2) By overriding the format of all tables using the `tabledata_format` keyword argument when writing out a VOTable file:

```
votable.to_xml("new_votable.xml", tabledata_format="binary")
```

Converting to/from an `astropy.table.Table`

The VOTable standard does not map conceptually to an **`astropy.table.Table`**. However, a single table within the `VOTable` file may be converted to and from an **`astropy.table.Table`**:

```
from astropy.io.votable import parse_single_table
table = parse_single_table("votable.xml").to_table()
```

As a convenience, there is also a function to create an entire VOTable file with just a single table:

```
from astropy.io.votable import from_table, writeto
votable = from_table(table)
writeto(votable, "output.xml")
```

Note

By default, `to_table` will use the `ID` attribute from the files to create the column names for the **`Table`** object. However, it may be that you want to use the `name` attributes instead. For this, set the `use_names_over_ids` keyword to **`True`**. Note that since field `names` are not guaranteed to be unique in the VOTable specification, but column names are required to be unique in `numpy` structured arrays (and thus **`astropy.table.Table`** objects), the names may be renamed by appending numbers to the end in some cases.

Performance Considerations

File reads will be moderately faster if the `TABLE` element includes an nrows attribute. If the number of rows is not specified, the record array must be resized repeatedly during load.

See Also

- VOTable Format Definition Version 1.1
- VOTable Format Definition Version 1.2
- VOTable Format Definition Version 1.3, Proposed Recommendation

Reference/API

astropy.io.votable Package

This package reads and writes data formats used by the Virtual Observatory (VO) initiative, particularly the VOTable XML format.

Functions

`parse`(source[, columns, invalid, verify, …])	Parses a VOTABLE xml file (or file-like object), and returns a `VOTableFile` object.
`parse_single_table`(source, **kwargs)	Parses a VOTABLE xml file (or file-like object), reading and returning only the first `Table` instance.
`validate`(source[, output, xmllint, filename])	Prints a validation report for the given file.
`from_table`(table[, table_id])	Given an `Table` object, return a `VOTableFile` file structure containing just that single table.
`is_votable`(source)	Reads the header of a file to determine if it is a VOTable file.
`writeto`(table, file[, tabledata_format])	Writes a `VOTableFile` to a VOTABLE xml file.

Classes

`Conf`()	Configuration parameters for `astropy.io.votable`.

astropy.io.votable.tree Module

Classes

`Link`([ID, title, value, href, action, id, …])	LINK elements: used to reference external documents and servers through a URI.
`Info`([ID, name, value, id, xtype, ref, …])	INFO elements: arbitrary key-value pairs for extensions to the standard.
`Values`(votable, field[, ID, null, ref, …])	VALUES element: used within FIELD and PARAM elements to define the domain of values.
`Field`(votable[, ID, name, datatype, …])	FIELD element: describes the datatype of a particular column of data.
`Param`(votable[, ID, name, value, datatype, …])	PARAM element: constant-valued columns in the data.
`CooSys`([ID, equinox, epoch, system, id, …])	COOSYS element: defines a coordinate system.
`TimeSys`([ID, timeorigin, timescale, …])	TIMESYS element: defines a time system.
`FieldRef`(table, ref[, ucd, utype, config, pos])	FIELDref element: used inside of GROUP elements to refer to remote FIELD elements.
`ParamRef`(table, ref[, ucd, utype, config, pos])	PARAMref element: used inside of GROUP elements to refer to remote PARAM elements.
`Group`(table[, ID, name, ref, ucd, utype, …])	GROUP element: groups FIELD and PARAM elements.
`Table`(votable[, ID, name, ref, ucd, utype, …])	TABLE element: optionally contains data.
`Resource`([name, ID, utype, type, id, …])	RESOURCE element: Groups TABLE and RESOURCE elements.
`VOTableFile`([ID, id, config, pos, version])	VOTABLE element: represents an entire file.

astropy.io.votable.converters Module

This module handles the conversion of various VOTABLE datatypes to/from TABLEDATA and BINARY formats.

Functions

`get_converter`(field[, config, pos])	Get an appropriate converter instance for a given field.
`table_column_to_votable_datatype`(column)	Given a **`astropy.table.Column`** instance, returns the attributes necessary to create a VOTable FIELD element that corresponds to the type of the column.

Classes

`Converter`(field[, config, pos])	The base class for all converters.

astropy.io.votable.ucd Module

This file contains routines to verify the correctness of UCD strings.

Functions

`parse_ucd`(ucd[, …])	Parse the UCD into its component parts.
`check_ucd`(ucd[, …])	Returns False if *ucd* is not a valid unified content descriptor.

astropy.io.votable.util Module

Various utilities and cookbook-like things.

Functions

`convert_to_writable_filelike`(fd[, compressed])	Returns a writable file-like object suitable for streaming output.
`coerce_range_list_param`(p[, frames, numeric])	Coerces and/or verifies the object *p* into a valid range-list-format parameter.

astropy.io.votable.validator Package

Validates a large collection of web-accessible VOTable files, and generates a report as a directory tree of HTML files.

Functions

`make_validation_report`([urls, destdir, …])	Validates a large collection of web-accessible VOTable files.

astropy.io.votable.xmlutil Module

Various XML-related utilities

Functions

`check_id`(ID[, name, config, pos])	Raises a `VOTableSpecError` if *ID* is not a valid XML ID.
`fix_id`(ID[, config, pos])	Given an arbitrary string, create one that can be used as an xml id.

check_token(token, attr_name[, config, pos])	Raises a ValueError if *token* is not a valid XML token.
check_mime_content_type(content_type[, …])	Raises a VOTableSpecError if *content_type* is not a valid MIME content type.
check_anyuri(uri[, config, pos])	Raises a VOTableSpecError if *uri* is not a valid URI.
validate_schema(filename[, version])	Validates the given file against the appropriate VOTable schema.

astropy.io.votable.exceptions Module

astropy.io.votable.exceptions

Contents

Warnings

Note

Most of the following warnings indicate violations of the VOTable specification. They should be reported to the authors of the tools that produced the VOTable file.

To control the warnings emitted, use the standard Python **`warnings`** module and the `astropy.io.votable.exceptions.conf.max_warnings` configuration item. Most of these are of the type **`VOTableSpecWarning`**.

W01: Array uses commas rather than whitespace

The VOTable spec states:

> If a cell contains an array or complex number, it should be encoded as multiple numbers separated by whitespace.

Many VOTable files in the wild use commas as a separator instead, and `astropy.io.votable` supports this convention when not in Pedantic mode.

`astropy.io.votable` always outputs files using only spaces, regardless of how they were input.

References: 1.1, 1.2

W02: x attribute 'y' is invalid. Must be a standard XML id

XML ids must match the following regular expression:

```
^[A-Za-z_][A-Za-z0-9_\.\-]*$
```

The VOTable 1.1 says the following:

> According to the XML standard, the attribute `ID` is a string beginning with a letter or underscore (`_`), followed by a sequence of letters, digits, or any of the punctuation characters `.` (dot), `-` (dash), `_` (underscore), or `:` (colon).

However, this is in conflict with the XML standard, which says colons may not be used. VOTable 1.1's own schema does not allow a colon here. Therefore, `astropy.io.votable` disallows the colon.

VOTable 1.2 corrects this error in the specification.

References: 1.1, XML Names

W03: Implicitly generating an ID from a name 'x' -> 'y'

The VOTable 1.1 spec says the following about `name` vs. `ID` on `FIELD` and `VALUE` elements:

> `ID` and `name` attributes have a different role in VOTable: the `ID` is meant as a *unique identifier* of an element seen as a VOTable component, while the `name` is meant for presentation purposes, and need not to be unique throughout the VOTable document. The `ID` attribute is therefore required in the elements which have to be referenced, but in principle any element may have an `ID` attribute. … In summary, the `ID` is different from the `name` attribute in that (a) the `ID` attribute is made from a restricted character set, and must be unique throughout a VOTable document whereas names are standard XML attributes and need not be unique; and (b) there should be support in the parsing software to look up references and extract the relevant element with matching `ID`.

It is further recommended in the VOTable 1.2 spec:

> While the `ID` attribute has to be unique in a VOTable document, the `name` attribute need not. It is however recommended, as a good practice, to assign unique names within a `TABLE` element. This recommendation means that, between a `TABLE` and its

corresponding closing `TABLE` tag, `name` attributes of `FIELD`, `PARAM` and optional `GROUP` elements should be all different.

Since `astropy.io.votable` requires a unique identifier for each of its columns, `ID` is used for the column name when present. However, when `ID` is not present, (since it is not required by the specification) `name` is used instead. However, `name` must be cleansed by replacing invalid characters (such as whitespace) with underscores.

Note

This warning does not indicate that the input file is invalid with respect to the VOTable specification, only that the column names in the record array may not match exactly the `name` attributes specified in the file.

References: 1.1, 1.2

W04: content-type 'x' must be a valid MIME content type

The `content-type` attribute must use MIME content-type syntax as defined in RFC 2046.

The current check for validity is somewhat over-permissive.

References: 1.1, 1.2

W05: 'x' is not a valid URI

The attribute must be a valid URI as defined in RFC 2396.

W06: Invalid UCD 'x': explanation

This warning is emitted when a `ucd` attribute does not match the syntax of a unified content descriptor.

If the VOTable version is 1.2 or later, the UCD will also be checked to ensure it conforms to the controlled vocabulary defined by UCD1+.

References: 1.1, 1.2

W07: Invalid astroYear in x: 'y'

As astro year field is a Besselian or Julian year matching the regular expression:

```
^[JB]?[0-9]+([.][0-9]*)?$
```

Defined in this XML Schema snippet:

```
<xs:simpleType  name="astroYear">
  <xs:restriction base="xs:token">
    <xs:pattern  value="[JB]?[0-9]+([.][0-9]*)?"/>
  </xs:restriction>
</xs:simpleType>
```

W08: ‘x’ must be a str or bytes object

To avoid local-dependent number parsing differences, `astropy.io.votable` may require a string or unicode string where a numeric type may make more sense.

W09: ID attribute not capitalized

The VOTable specification uses the attribute name `ID` (with uppercase letters) to specify unique identifiers. Some VOTable-producing tools use the more standard lowercase `id` instead. `astropy.io.votable` accepts `id` and emits this warning if `verify` is `'warn'`.

References: 1.1, 1.2

W10: Unknown tag ‘x’. Ignoring

The parser has encountered an element that does not exist in the specification, or appears in an invalid context. Check the file against the VOTable schema (with a tool such as xmllint. If the file validates against the schema, and you still receive this warning, this may indicate a bug in `astropy.io.votable`.

References: 1.1, 1.2

W11: The gref attribute on LINK is deprecated in VOTable 1.1

Earlier versions of the VOTable specification used a `gref` attribute on the `LINK` element to specify a GLU reference. New files should specify a `glu:` protocol using the `href` attribute.

Since `astropy.io.votable` does not currently support GLU references, it likewise does not automatically convert the `gref` attribute to the new form.

References: 1.1, 1.2

W12: ‘x’ element must have at least one of ‘ID’ or ‘name’ attributes

In order to name the columns of the Numpy record array, each `FIELD` element must have either an `ID` or `name` attribute to derive a name from. Strictly speaking, according to the VOTable schema, the `name` attribute is required. However, if `name` is not present by `ID` is, and `verify` is not `'exception'`, `astropy.io.votable` will continue without a `name` defined.

References: 1.1, 1.2

W13: ‘x’ is not a valid VOTable datatype, should be ‘y’

Some VOTable files in the wild use non-standard datatype names. These are mapped to standard ones using the following mapping:

```
string        -> char
unicodeString -> unicodeChar
int16         -> short
int32         -> int
```

```
int64         -> long
float32       -> float
float64       -> double
unsignedInt   -> long
unsignedShort -> int
```

To add more datatype mappings during parsing, use the `datatype_mapping` keyword to `astropy.io.votable.parse`.

References: 1.1, 1.2

W15: x element missing required 'name' attribute

The `name` attribute is required on every `FIELD` element. However, many VOTable files in the wild omit it and provide only an `ID` instead. In this case, when `verify` is not `'exception'` `astropy.io.votable` will copy the `name` attribute to a new `ID` attribute.

References: 1.1, 1.2

W17: x element contains more than one DESCRIPTION element

A `DESCRIPTION` element can only appear once within its parent element.

According to the schema, it may only occur once (1.1, 1.2)

However, it is a proposed extension to VOTable 1.2.

W18: TABLE specified nrows=x, but table contains y rows

The number of rows explicitly specified in the `nrows` attribute does not match the actual number of rows (`TR` elements) present in the `TABLE`. This may indicate truncation of the file, or an internal error in the tool that produced it. If `verify` is not `'exception'`, parsing will proceed, with the loss of some performance.

References: 1.1, 1.2

W19: The fields defined in the VOTable do not match those in the embedded FITS file

The column fields as defined using `FIELD` elements do not match those in the headers of the embedded FITS file. If `verify` is not `'exception'`, the embedded FITS file will take precedence.

W20: No version number specified in file. Assuming 1.1

If no version number is explicitly given in the VOTable file, the parser assumes it is written to the VOTable 1.1 specification.

W21: astropy.io.votable is designed for VOTable version 1.1, 1.2, 1.3, and 1.4, but this file is x

Unknown issues may arise using `astropy.io.votable` with VOTable files from a version other than 1.1, 1.2, 1.3, or 1.4.

W22: The DEFINITIONS element is deprecated in VOTable 1.1. Ignoring

Version 1.0 of the VOTable specification used the `DEFINITIONS` element to define coordinate systems. Version 1.1 now uses `COOSYS` elements throughout the document.

References: 1.1, 1.2

W23: Unable to update service information for 'x'

Raised when the VO service database can not be updated (possibly due to a network outage). This is only a warning, since an older and possible out-of-date VO service database was available locally.

W24: The VO catalog database is for a later version of astropy.io.votable

The VO catalog database retrieved from the www is designed for a newer version of `astropy.io.votable`. This may cause problems or limited features performing service queries. Consider upgrading `astropy.io.votable` to the latest version.

W25: 'service' failed with: ...

A VO service query failed due to a network error or malformed arguments. Another alternative service may be attempted. If all services fail, an exception will be raised.

W26: 'child' inside 'parent' added in VOTable X.X

The given element was not supported inside of the given element until the specified VOTable version, however the version declared in the file is for an earlier version. These attributes may not be written out to the file.

W27: COOSYS deprecated in VOTable 1.2

The `COOSYS` element was deprecated in VOTABLE version 1.2 in favor of a reference to the Space-Time Coordinate (STC) data model (see utype and the IVOA note referencing STC in VOTable.

W28: 'attribute' on 'element' added in VOTable X.X

The given attribute was not supported on the given element until the specified VOTable version, however the version declared in the file is for an earlier version. These attributes may not be written out to the file.

W29: Version specified in non-standard form 'v1.0'

Some VOTable files specify their version number in the form "v1.0", when the only supported forms in the spec are "1.0".

References: 1.1, 1.2

W30: Invalid literal for float 'x'. Treating as empty.

Some VOTable files write missing floating-point values in non-standard ways, such as "null" and "-". If `verify` is not `'exception'`, any non-standard floating-point literals are treated as missing values.

References: 1.1, 1.2

W31: NaN given in an integral field without a specified null value

Since NaN's can not be represented in integer fields directly, a null value must be specified in the FIELD descriptor to support reading NaN's from the tabledata.

References: 1.1, 1.2

W32: Duplicate ID 'x' renamed to 'x_2' to ensure uniqueness

Each field in a table must have a unique ID. If two or more fields have the same ID, some will be renamed to ensure that all IDs are unique.

From the VOTable 1.2 spec:

> The `ID` and `ref` attributes are defined as XML types `ID` and `IDREF` respectively. This means that the contents of `ID` is an identifier which must be unique throughout a VOTable document, and that the contents of the `ref` attribute represents a reference to an identifier which must exist in the VOTable document.

References: 1.1, 1.2

W33: Column name 'x' renamed to 'x_2' to ensure uniqueness

Each field in a table must have a unique name. If two or more fields have the same name, some will be renamed to ensure that all names are unique.

References: 1.1, 1.2

W34: 'x' is an invalid token for attribute 'y'

The attribute requires the value to be a valid XML token, as defined by XML 1.0.

W35: 'x' attribute required for INFO elements

The `name` and `value` attributes are required on all `INFO` elements.

References: 1.1, 1.2

W36: null value 'x' does not match field datatype, setting to 0

If the field specifies a `null` value, that value must conform to the given `datatype`.

References: 1.1, 1.2

W37: Unsupported data format 'x'

The 3 datatypes defined in the VOTable specification and supported by `astropy.io.votable` are `TABLEDATA`, `BINARY` and `FITS`.

References: 1.1, 1.2

W38: Inline binary data must be base64 encoded, got 'x'

The only encoding for local binary data supported by the VOTable specification is base64.

W39: Bit values can not be masked

Bit values do not support masking. This warning is raised upon setting masked data in a bit column.

References: 1.1, 1.2

W40: 'cprojection' datatype repaired

This is a terrible hack to support Simple Image Access Protocol results from archive.noao.edu. It creates a field for the coordinate projection type of type "double", which actually contains character data. We have to hack the field to store character data, or we can't read it in. A warning will be raised when this happens.

W41: An XML namespace is specified, but is incorrect. Expected 'x', got 'y'

An XML namespace was specified on the `VOTABLE` element, but the namespace does not match what is expected for a `VOTABLE` file.

The `VOTABLE` namespace is:

```
http://www.ivoa.net/xml/VOTable/vX.X
```

where "X.X" is the version number.

Some files in the wild set the namespace to the location of the VOTable schema, which is not correct and will not pass some validating parsers.

W42: No XML namespace specified

The root element should specify a namespace.

The `VOTABLE` namespace is:

```
http://www.ivoa.net/xml/VOTable/vX.X
```

where "X.X" is the version number.

W43: element ref='x' which has not already been defined

Referenced elements should be defined before referees. From the VOTable 1.2 spec:

> In VOTable1.2, it is further recommended to place the ID attribute prior to referencing it whenever possible.

W44: VALUES element with ref attribute has content ('element')

`VALUES` elements that reference another element should not have their own content.

From the VOTable 1.2 spec:

> The `ref` attribute of a `VALUES` element can be used to avoid a

repetition of the domain definition, by referring to a previously defined `VALUES` element having the referenced `ID` attribute. When specified, the `ref` attribute defines completely the domain without any other element or attribute, as e.g. `<VALUES ref="RAdomain"/>`

W45: content-role attribute ‘x’ invalid

The `content-role` attribute on the `LINK` element must be one of the following:

```
query, hints, doc, location
```

And in VOTable 1.3, additionally:

```
type
```

References: 1.1, 1.2 1.3

W46: char or unicode value is too long for specified length of x

The given char or unicode string is too long for the specified field length.

W47: Missing arraysize indicates length 1

If no arraysize is specified on a char field, the default of ‘1’ is implied, but this is rarely what is intended.

W48: Unknown attribute ‘attribute’ on element

The attribute is not defined in the specification.

W49: Empty cell illegal for integer fields.

Prior to VOTable 1.3, the empty cell was illegal for integer fields.

If a “null” value was specified for the cell, it will be used for the value, otherwise, 0 will be used.

W50: Invalid unit string ‘x’

Invalid unit string as defined in the Standards for Astronomical Catalogues, Version 2.0.

Consider passing an explicit `unit_format` parameter if the units in this file conform to another specification.

W51: Value ‘x’ is out of range for a n-bit integer field

The integer value is out of range for the size of the field.

W52: The BINARY2 format was introduced in VOTable 1.3, but this file is declared as version ‘1.2’

The BINARY2 format was introduced in VOTable 1.3. It should not be present in files marked as an earlier version.

W53: VOTABLE element must contain at least one RESOURCE element.

The VOTABLE element must contain at least one RESOURCE element.

W54: The TIMESYS element was introduced in VOTable 1.4, but this file is declared as version '1.3'

The TIMESYS element was introduced in VOTable 1.4. It should not be present in files marked as an earlier version.

W55: FIELD () has datatype="char" but contains non-ASCII value ()

When non-ASCII characters are detected when reading a TABLEDATA value for a FIELD with `datatype="char"`, we can issue this warning.

Exceptions

Note

This is a list of many of the fatal exceptions emitted by `astropy.io.votable` when the file does not conform to spec. Other exceptions may be raised due to unforeseen cases or bugs in `astropy.io.votable` itself.

E01: Invalid size specifier 'x' for a char/unicode field (in field 'y')

The size specifier for a `char` or `unicode` field must be only a number followed, optionally, by an asterisk. Multi-dimensional size specifiers are not supported for these datatypes.

Strings, which are defined as a set of characters, can be represented in VOTable as a fixed- or variable-length array of characters:

```
<FIELD name="unboundedString" datatype="char" arraysize="*"/>
```

A 1D array of strings can be represented as a 2D array of characters, but given the logic above, it is possible to define a variable-length array of fixed-length strings, but not a fixed-length array of variable-length strings.

E02: Incorrect number of elements in array. Expected multiple of x, got y

The number of array elements in the data does not match that specified in the FIELD specifier.

E03: 'x' does not parse as a complex number

Complex numbers should be two values separated by whitespace.

References: 1.1, 1.2

E04: Invalid bit value 'x'

A `bit` array should be a string of '0's and '1's.

References: 1.1, 1.2

E05: Invalid boolean value 'x'

A `boolean` value should be one of the following strings (case insensitive) in the `TABLEDATA` format:

```
'TRUE', 'FALSE', '1', '0', 'T', 'F', '\0', ' ', '?'
```

and in `BINARY` format:

```
'T', 'F', '1', '0', '\0', ' ', '?'
```

References: 1.1, 1.2

E06: Unknown datatype 'x' on field 'y'

The supported datatypes are:

```
double, float, bit, boolean, unsignedByte, short, int, long,
floatComplex, doubleComplex, char, unicodeChar
```

The following non-standard aliases are also supported, but in these case W13 will be raised:

```
string        -> char
unicodeString -> unicodeChar
int16         -> short
int32         -> int
int64         -> long
float32       -> float
float64       -> double
unsignedInt   -> long
unsignedShort -> int
```

To add more datatype mappings during parsing, use the `datatype_mapping` keyword to **`astropy.io.votable.parse`**.

References: 1.1, 1.2

E08: type must be 'legal' or 'actual', but is 'x'

The `type` attribute on the `VALUES` element must be either `legal` or `actual`.

References: 1.1, 1.2

E09: 'x' must have a value attribute

The `MIN`, `MAX` and `OPTION` elements must always have a `value` attribute.

References: 1.1, 1.2

E10: 'datatype' attribute required on all 'FIELD' elements

From VOTable 1.1 and later, `FIELD` and `PARAM` elements must have a `datatype` field.

References: 1.1, 1.2

E11: precision 'x' is invalid

The precision attribute is meant to express the number of significant digits, either as a number of decimal places (e.g. `precision="F2"` or equivalently `precision="2"` to express 2 significant figures after the decimal point), or as a number of significant figures (e.g. `precision="E5"` indicates a relative precision of 10-5).

It is validated using the following regular expression:

```
[EF]?[1-9][0-9]*
```

References: 1.1, 1.2

E12: width must be a positive integer, got 'x'

The width attribute is meant to indicate to the application the number of characters to be used for input or output of the quantity.

References: 1.1, 1.2

E13: Invalid arraysize attribute 'x'

From the VOTable 1.2 spec:

> A table cell can contain an array of a given primitive type, with a fixed or variable number of elements; the array may even be multidimensional. For instance, the position of a point in a 3D space can be defined by the following:
>
> ```
> <FIELD ID="point_3D" datatype="double" arraysize="3"/>
> ```
>
> and each cell corresponding to that definition must contain exactly 3 numbers. An asterisk (*) may be appended to indicate a variable number of elements in the array, as in:
>
> ```
> <FIELD ID="values" datatype="int" arraysize="100*"/>
> ```
>
> where it is specified that each cell corresponding to that definition contains 0 to 100 integer numbers. The number may be omitted to specify an unbounded array (in practice up to =~2×10^9 elements).
>
> A table cell can also contain a multidimensional array of a given primitive type. This is specified by a sequence of dimensions separated by the `x` character, with the first dimension changing fastest; as in the

case of a simple array, the last dimension may be variable in length. As an example, the following definition declares a table cell which may contain a set of up to 10 images, each of 64×64 bytes:

```
<FIELD ID="thumbs" datatype="unsignedByte"
arraysize="64×64×10*"/>
```

References: 1.1, 1.2

E14: value attribute is required for all PARAM elements

All `PARAM` elements must have a `value` attribute.

References: 1.1, 1.2

E15: ID attribute is required for all COOSYS elements

All `COOSYS` elements must have an `ID` attribute.

Note that the VOTable 1.1 specification says this attribute is optional, but its corresponding schema indicates it is required.

In VOTable 1.2, the `COOSYS` element is deprecated.

E16: Invalid system attribute 'x'

The `system` attribute on the `COOSYS` element must be one of the following:

```
'eq_FK4', 'eq_FK5', 'ICRS', 'ecl_FK4', 'ecl_FK5', 'galactic',
'supergalactic', 'xy', 'barycentric', 'geo_app'
```

References: 1.1

E17: extnum must be a positive integer

`extnum` attribute must be a positive integer.

References: 1.1, 1.2

E18: type must be 'results' or 'meta', not 'x'

The `type` attribute of the `RESOURCE` element must be one of "results" or "meta".

References: 1.1, 1.2

E19: File does not appear to be a VOTABLE

Raised either when the file doesn't appear to be XML, or the root element is not VOTABLE.

E20: Data has more columns than are defined in the header (x)

The table had only *x* fields defined, but the data itself has more columns than that.

E21: Data has fewer columns (x) than are defined in the header (y)

The table had *x* fields defined, but the data itself has only *y* columns.

E22: ID attribute is required for all TIMESYS elements

All `TIMESYS` elements must have an `ID` attribute.

E23: Invalid timeorigin attribute 'x'

The `timeorigin` attribute on the `TIMESYS` element must be either a floating point literal specifiying a valid Julian Date, or, for convenience, the string "MJD-origin" (standing for 2400000.5) or the string "JD-origin" (standing for 0).

References: 1.4

E24: Attempt to write non-ASCII value () to FIELD () which has datatype="char"

Non-ASCII unicode values should not be written when the FIELD `datatype="char"`, and cannot be written in BINARY or BINARY2 serialization.

E25: No FIELDs are defined; DATA section will be ignored.

A VOTable cannot have a DATA section without any defined FIELD; DATA will be ignored.

Exception Utilities

class `astropy.io.votable.exceptions.`**`Conf`**

Configuration parameters for `astropy.io.votable.exceptions`.

`max_warnings`

Number of times the same type of warning is displayed before being suppressed

`astropy.io.votable.exceptions.`**`warn_or_raise`**(*warning_class*, *exception_class=None*, *args=()*, *config=None*, *pos=None*, *stacklevel=1*)

Warn or raise an exception, depending on the verify setting.

`astropy.io.votable.exceptions.`**`vo_raise`**(*exception_class*, *args=()*, *config=None*, *pos=None*)

Raise an exception, with proper position information if available.

`astropy.io.votable.exceptions.`**`vo_reraise`**(*exc*, *config=None*, *pos=None*, *additional=''*)

Raise an exception, with proper position information if available.

Restores the original traceback of the exception, and should only be called within an "except:" block of code.

`astropy.io.votable.exceptions.`**`vo_warn`**(*warning_class*, *args=()*, *config=None*, *pos=None*, *stacklevel=1*)

Warn, with proper position information if available.

`astropy.io.votable.exceptions.` **`parse_vowarning`**(*line*)

Parses the vo warning string back into its parts.

class `astropy.io.votable.exceptions.` **`VOWarning`**(*args*, *config=None*, *pos=None*)

Bases: **`astropy.utils.exceptions.AstropyWarning`**

The base class of all VO warnings and exceptions.

Handles the formatting of the message with a warning or exception code, filename, line and column number.

class `astropy.io.votable.exceptions.` **`VOTableChangeWarning`**(*args*, *config=None*, *pos=None*)

Bases: **`astropy.io.votable.exceptions.VOWarning`**, **`SyntaxWarning`**

A change has been made to the input XML file.

class `astropy.io.votable.exceptions.` **`VOTableSpecWarning`**(*args*, *config=None*, *pos=None*)

Bases: **`astropy.io.votable.exceptions.VOWarning`**, **`SyntaxWarning`**

The input XML file violates the spec, but there is an obvious workaround.

class `astropy.io.votable.exceptions.` **`UnimplementedWarning`**(*args*, *config=None*, *pos=None*)

Bases: **`astropy.io.votable.exceptions.VOWarning`**, **`SyntaxWarning`**

A feature of the VOTABLE spec is not implemented.

class `astropy.io.votable.exceptions.` **`IOWarning`**(*args*, *config=None*, *pos=None*)

Bases: **`astropy.io.votable.exceptions.VOWarning`**, **`RuntimeWarning`**

A network or IO error occurred, but was recovered using the cache.

class `astropy.io.votable.exceptions.` **`VOTableSpecError`**(*args*, *config=None*, *pos=None*)

Bases: **`astropy.io.votable.exceptions.VOWarning`**, **`ValueError`**

The input XML file violates the spec and there is no good workaround.

Miscellaneous: HDF5, YAML, ASDF, pickle

(`astropy.io.misc`)

The `astropy.io.misc` module contains miscellaneous input/output routines that do not fit elsewhere, and are often used by other `astropy` sub-packages. For example, `astropy.io.misc.hdf5` contains functions to read/write `Table` objects from/to HDF5 files, but these should not be imported directly by users. Instead, users can access this functionality via the `Table` class itself (see Unified File Read/Write Interface). Routines that are intended to be used directly by users are listed in the `astropy.io.misc` section.

astropy.io.misc Package

This package contains miscellaneous utility functions for data input/output with astropy.

Functions

`fnpickle`(object, fileorname[, protocol, append])	Pickle an object to a specified file.
`fnunpickle`(fileorname[, number])	Unpickle pickled objects from a specified file and return the contents.

astropy.io.misc.hdf5 Module

This package contains functions for reading and writing HDF5 tables that are not meant to be used directly, but instead are available as readers/writers in `astropy.table`. See Unified File Read/Write Interface for more details.

Functions

`read_table_hdf5`(input[, path, …])	Read a Table object from an HDF5 file
`write_table_hdf5`(table, output[, path, …])	Write a Table object to an HDF5 file

astropy.io.misc.yaml Module

This module contains functions for serializing core astropy objects via the YAML protocol. It provides functions `dump`, `load`, and `load_all` which call the corresponding functions in PyYaml but use the `AstropyDumper` and `AstropyLoader` classes to define custom YAML tags for the following astropy classes: - `astropy.units.Unit` - `astropy.units.Quantity` - `astropy.time.Time` - `astropy.time.TimeDelta` - `astropy.coordinates.SkyCoord` - `astropy.coordinates.Angle` - `astropy.coordinates.Latitude` - `astropy.coordinates.Longitude` - `astropy.coordinates.EarthLocation` - `astropy.table.SerializedColumn`

Note

This module requires PyYaml version 3.13 or later.

Example

::

```
>>> from astropy.io.misc import yaml
>>> import astropy.units as u
>>> from astropy.time import Time
>>> from astropy.coordinates import EarthLocation
>>> t = Time(2457389.0, format='mjd',
...          location=EarthLocation(1000, 2000, 3000, unit=u.km))
>>> td = yaml.dump(t)
>>> print(td)
!astropy.time.Time
format: mjd
in_subfmt: '*'
jd1: 4857390.0
jd2: -0.5
location: !astropy.coordinates.earth.EarthLocation
  ellipsoid: WGS84
  x: !astropy.units.Quantity
    unit: &id001 !astropy.units.Unit {unit: km}
    value: 1000.0
  y: !astropy.units.Quantity
    unit: *id001
    value: 2000.0
  z: !astropy.units.Quantity
    unit: *id001
    value: 3000.0
out_subfmt: '*'
precision: 3
scale: utc
>>> ty = yaml.load(td)
>>> ty
<Time object: scale='utc' format='mjd' value=2457389.0>
>>> ty.location
<EarthLocation (1000., 2000., 3000.) km>
```

Functions

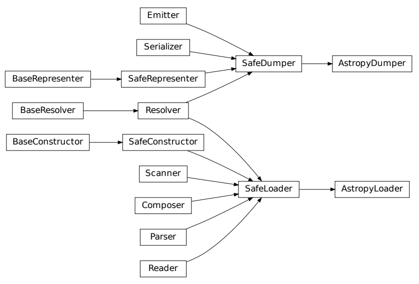
`load`(stream)	Parse the first YAML document in a stream using the AstropyLoader and produce the corresponding Python object.
`load_all`(stream)	Parse the all YAML documents in a stream using the AstropyLoader class and produce the corresponding Python object.

dump(data[, stream])	Serialize a Python object into a YAML stream using the AstropyDumper class.

Classes

AstropyLoader(stream)	Custom SafeLoader that constructs astropy core objects as well as Python tuple and unicode objects.
AstropyDumper(stream[, default_style, …])	Custom SafeDumper that represents astropy core objects as well as Python tuple and unicode objects.

Class Inheritance Diagram



astropy.io.misc.asdf Package

The **asdf** sub-package contains code that is used to serialize `astropy` types so that they can be represented and stored using the Advanced Scientific Data Format (ASDF).

If both **asdf** and **astropy** are installed, no further configuration is required in order to process ASDF files that contain **astropy** types. The **asdf** package has been designed to automatically detect the presence of the tags defined by **astropy**.

For convenience, users can write **Table** objects to ASDF files using the Unified File Read/Write Interface. See Using ASDF With Table I/O below.

Documentation on the ASDF Standard can be found here. Documentation on the ASDF Python module can be found here. Additional details for Astropy developers can be found in Details.

Using ASDF With Table I/O

ASDF provides readers and writers for **Table** using the Unified File Read/Write Interface. This makes it convenient to read and write ASDF files with **Table** data.

Basic Usage

Given a table, it is possible to write it out to an ASDF file:

```
from astropy.table import Table

# Create a simple table
t = Table(dtype=[('a', 'f4'), ('b', 'i4'), ('c', 'S2')])
# Write the table to an ASDF file
t.write('table.asdf')
```

The I/O registry automatically selects the appropriate writer function to use based on the `.asdf` extension of the output file.

Reading a file generated in this way is also possible using **`read`**:

```
t2 = Table.read('table.asdf')
```

The I/O registry automatically selects the appropriate reader function based on the extension of the input file.

In the case of both reading and writing, if the file extension is not `.asdf` it is possible to explicitly specify the reader/writer function to be used:

```
t3 = Table.read('table.zxcv', format='asdf')
```

Advanced Usage

The fundamental ASDF data structure is the tree, which is a nested combination of basic data structures (see this for a more detailed description). At the top level, the tree is a **`dict`**.

The consequence of this is that a **`Table`** object (or any object, for that matter) can be stored at any arbitrary location within an ASDF tree. The basic writer use case described above stores the given **`Table`** at the top of the tree using a default key. The basic reader case assumes that a **`Table`** is stored in the same place.

However, it may sometimes be useful for users to specify a different top-level key to be used for storage and retrieval of a **`Table`** from an ASDF file. For this reason, the ASDF I/O interface provides `data_key` as an optional keyword when writing and reading:

```
from astropy.table import Table

t = Table(dtype=[('a', 'f4'), ('b', 'i4'), ('c', 'S2')])
# Write the table to an ASDF file using a non-default key
```

```
t.write('foo.asdf', data_key='foo')
```

A **Table** stored using a custom data key can be retrieved by passing the same argument to **read**:

```
foo = Table.read('foo.asdf', data_key='foo')
```

The `data_key` option only applies to **Table** objects that are stored at the top of the ASDF tree. For full generality, users may pass a callback when writing or reading ASDF files to define precisely where the **Table** object should be placed in the tree. The option for the write case is `make_tree`. The function callback should accept exactly one argument, which is the **Table** object, and should return a **dict** representing the tree to be stored:

```
def make_custom_tree(table):
    # Return a nested tree where the table is stored at the second
level
    return dict(foo=dict(bar=table))

t = Table(dtype=[('a', 'f4'), ('b', 'i4'), ('c', 'S2')])
# Write the table to an ASDF file using a non-default key
t.write('foobar.asdf', make_tree=make_custom_tree)
```

Similarly, when reading an ASDF file, the user can pass a custom callback to locate the table within the ASDF tree. The option in this case is `find_table`. The callback should accept exactly one argument, which is an **dict** representing the ASDF tree, and it should return a **Table** object:

```
def find_table(tree):
    # This returns the Table that was stored by the example above
    return tree['foo']['bar']

foo = Table.read('foobar.asdf', find_table=find_table)
```

Details

The **asdf** sub-package defines classes, referred to as **tags**, that implement the logic for serialization and deserialization of `astropy` types. Users should never need to refer to tag implementations directly. Their presence should be entirely transparent when processing ASDF files.

ASDF makes use of abstract data type definitions called **schemas**. The tag classes provided here are specific implementations of particular schemas. Some of the tags in `astropy` (e.g., those related to transforms) implement schemas that are defined by the ASDF Standard. In other cases, both the tags and schemas are defined within `astropy` (e.g., those related to many of the coordinate frames). Documentation of the individual schemas defined by `astropy` can be found below in the Schemas section.

Not all `astropy` types are currently serializable by ASDF. Attempting to write unsupported types to an ASDF file will lead to a `RepresenterError`. In order to support new types, new tags and schemas must be created. See Writing ASDF Extensions for additional details, as well as the following example.

Example: Adding a New Object to the Astropy ASDF Extension

In this example, we will show how to implement serialization for a new **`Model`** object, but the basic principles apply to serialization of other `astropy` objects. As mentioned, adding a new object to the `astropy` ASDF extension requires both a tag and a schema.

All schemas for transforms are currently defined within the ASDF standard. Any new serializable transforms must have a corresponding new schema here. Let's consider a new model called `MyModel`, a new model in `astropy.modeling.functional_models` that has two parameters `amplitude` and `x_0`. We would like to strictly require both of these parameters be set. We would also like to specify that these parameters can either be numeric type, or `astropy.units.quantity` type. A schema describing this model would look like:

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://stsci.edu/schemas/asdf/transform/mymodel-1.0.0"
tag: "tag:stsci.edu:asdf/transform/mymodel-1.0.0"
title: >
  Example new model.

description: >
  Example new model, which describes the distribution of ABC.

allOf:
  - $ref: "transform-1.2.0"
  - type: object
    properties:
      amplitude:
        anyOf:
          - $ref: "../unit/quantity-1.1.0"
          - type: number
        description: Amplitude of distribution.
      x_0:
        anyOf:
          - $ref: "../unit/quantity-1.1.0"
          - type: number
        description: X center position.

    required: ['amplitude', 'x_0]
...
```

All new transform schemas reference the base transform schema of the latest type. This schema describes the other model attributes that are common to all or many models, so that individual schemas only handle the parameters specific to that model. Additionally, this schema references the latest verison of the `quantity` schema, so that models can retain information about units and quantities. References allow previously defined objects to be used inside new custom types.

The next component is the tag class. This class must have a `to_tree` method in which the required attributes of the object in question are obtained, and a `from_tree` method which reconstructs the object based on the parameters written to the ASDF file. `astropy` Models inherit from the `TransformType` base class tag, which takes care of attributes (e.g `name`, `bounding_box`, `n_inputs`) that are common to all or many Model classes to limit redundancy in individual tags. Each individual model tag then only has to obtain and set model-specific parameters:

```
from .basic import TransformType
from . import _parameter_to_value

class MyModelType(TransformType):
name = 'transform/mymodel'
version = '1.0.0'
types = ['astropy.modeling.functional_models.MyModel']

@classmethod
def from_tree_transform(cls, node, ctx):
    return functional_models.MyModel(amplitude=node['amplitude'],
                                     x_0=node['x_0'])

@classmethod
def to_tree_transform(cls, model, ctx):
    node = {'amplitude': _parameter_to_value(amplitude),
            'x_0': _parameter_to_value(x_0)}
    return node
```

This tag class contains all the machinery to deconstruct objects to and reconstruct them from ASDF files. The tag class - by convention named by the object name appended with 'Type' - references the schema and version, and the object in `astropy.modeling.functional_models`. The basic model parameters are handled in the `to_tree_transform` and `from_tree_transform` of the base `TransformType` class, while model-specific parameters are handled here in `MyModelType`. Since this model can take units and quantities with input parameters, the imported``_parameter_to_value`` allows this to flexibly work with both basic numeric values as well as quantities.

Schemas

Documentation for each of the individual ASDF schemas defined by `astropy` can be found below.

Schemas

Documentation for each of the individual ASDF schemas defined by `astropy` can be found at the links below.

Documentation for the schemas defined in the ASDF Standard can be found here.

Contents

Coordinates

The following schemas are associated with `astropy` types from the Astronomical Coordinate Systems (astropy.coordinates) submodule:

coordinates/angle-1.0.0

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://astropy.org/schemas/astropy/coordinates/angle-1.0.0"
tag: "tag:astropy.org:astropy/coordinates/angle-1.0.0"

title: |
  Represents an Angle.

description:
  This object represents a subtype of Quantity which has units
equivalent to
  radians or degrees.

examples:
  -
    - An Angle object in Degrees
    - |
        !<tag:astropy.org:astropy/coordinates/angle-1.0.0>
          unit: !unit/unit-1.0.0 deg
          value: 10.0

type: object
properties:
  value:
    description: |
      A vector of one or more values
    anyOf:
      - type: number
      - $ref: "tag:stsci.edu:asdf/core/ndarray-1.0.0"
  unit:
    description: |
      The unit corresponding to the values
    $ref: "tag:stsci.edu:asdf/unit/unit-1.0.0"
required: [value, unit]
...
```

coordinates/earthlocation-1.0.0

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
```

```
id: "http://astropy.org/schemas/astropy/coordinates/earthlocation-
1.0.0"
tag: "tag:astropy.org:astropy/coordinates/earthlocation-1.0.0"

title: |
  Represents EarthLocation objects from astropy.

description: |
  Location on the Earth.

type: object
properties:
  x:
    description: |
      X component of location in geocentric representation
    $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
  y:
    description: |
      Y component of location in geocentric representation
    $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
  z:
    description: |
      Z component of location in geocentric representation
    $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
  ellipsoid:
    description: |
      Reference ellipsoid that is used when representing geodetic
coordinates.
    type: string
    enum: [WGS84, GRS80, WGS72]

required: [x, y, z]
additionalProperties: False
...
```

coordinates/latitude-1.0.0

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://astropy.org/schemas/astropy/coordinates/latitude-1.0.0"
tag: "tag:astropy.org:astropy/coordinates/latitude-1.0.0"

title: |
  Represents latitude-like angles.

description: |
  Represents latitude-like angle(s) which must be in the range -90 to
```

```
+90 deg.

examples:
  -
    - A Latitude object in Degrees
    - |
        !<tag:astropy.org:astropy/coordinates/latitude-1.0.0>
          unit: !unit/unit-1.0.0 deg
          value: 10.0

type: object
properties:
  value:
    description: |
      A vector of one or more values
    anyOf:
      - type: number
      - $ref: "tag:stsci.edu:asdf/core/ndarray-1.0.0"
  unit:
    description: |
      The unit corresponding to the values
    $ref: "tag:stsci.edu:asdf/unit/unit-1.0.0"
required: [value, unit]
...
```

coordinates/longitude-1.0.0

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://astropy.org/schemas/astropy/coordinates/longitude-1.0.0"
tag: "tag:astropy.org:astropy/coordinates/longitude-1.0.0"

title: |
  Represents longitude-like angles.

description: |
    Longitude-like angle(s) which are wrapped within a contiguous 360
degree range.

examples:
  -
    - A Longitude object in Degrees
    - |
        !<tag:astropy.org:astropy/coordinates/longitude-1.0.0>
          unit: !unit/unit-1.0.0 deg
          value: 10.0
          wrap_angle: !<tag:astropy.org:astropy/coordinates/angle-
```

```
1.0.0>
            unit: !unit/unit-1.0.0 deg
            value: 180.0

type: object
properties:
  value:
    description: |
      A vector of one or more values
    anyOf:
      - type: number
      - $ref: "tag:stsci.edu:asdf/core/ndarray-1.0.0"
  unit:
    description: |
      The unit corresponding to the values
    $ref: "tag:stsci.edu:asdf/unit/unit-1.0.0"
  wrap_angle:
    description: |
      Angle at which to wrap back to ``wrap_angle - 360 deg``.
    $ref: "angle-1.0.0"

required: [value, unit, wrap_angle]
...
```

coordinates/representation-1.0.0

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://astropy.org/schemas/astropy/coordinates/representation-
1.0.0"
tag: "tag:astropy.org:astropy/coordinates/representation-1.0.0"

title: |
  Representation of points or differentials in two or three
dimensional space.

description: |
  Representation of points or differentials in two or three
dimensional space.

examples:
  -
    - A SphericalRepresentation
    - |
        !<tag:astropy.org:astropy/coordinates/representation-1.0.0>
          components:
            distance: !unit/quantity-1.1.0 {unit: !unit/unit-1.0.0
AU, value: 1.0}
            lat: !<tag:astropy.org:astropy/coordinates/latitude-
1.0.0> {unit: !unit/unit-1.0.0 deg,
              value: 10.0}
            lon: !<tag:astropy.org:astropy/coordinates/longitude-
1.0.0>
              unit: !unit/unit-1.0.0 deg
              value: 10.0
              wrap_angle: !<tag:astropy.org:astropy/coordinates
/angle-1.0.0> {unit: !unit/unit-1.0.0 deg,
                value: 360.0}
          type: SphericalRepresentation
  -
    - A CartesianDifferential
    - |
      !<tag:astropy.org:astropy/coordinates/representation-1.0.0>
        components:
          d_x: !unit/quantity-1.1.0 {unit: !unit/unit-1.0.0 km s-1,
value: 100.0}
          d_y: !unit/quantity-1.1.0 {unit: !unit/unit-1.0.0 km s-1,
value: 200.0}
          d_z: !unit/quantity-1.1.0 {unit: !unit/unit-1.0.0 km s-1,
value: 3141.0}
          type: CartesianDifferential

type: object
properties:
  type:
    type: string
    enum:
      - CartesianRepresentation
      - SphericalRepresentation
      - UnitSphericalRepresentation
      - RadialRepresentation
      - PhysicsSphericalRepresentation
      - CylindricalRepresentation
      - CartesianDifferential
      - SphericalDifferential
      - UnitSphericalCosLatDifferential
      - UnitSphericalDifferential
      - SphericalCosLatDifferential
      - RadialDifferential
      - PhysicsSphericalDifferential
      - CylindricalDifferential

  components:
    anyOf:
      # CartesianRepresentation
      - type: object
        properties:
          x:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          y:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          z:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"

      # SphericalRepresentation
      - type: object
        properties:
          lat:
            $ref: "latitude-1.0.0"
          lon:
            $ref: "longitude-1.0.0"
          distance:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"

      # UnitSphericalRepresentation
      - type: object
        properties:
          lat:
            $ref: "latitude-1.0.0"
          lon:
            $ref: "longitude-1.0.0"

      # RadialRepresentation
      - type: object
        properties:
          distance:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"

      # PhysicsSphericalRepresentation
      - type: object
        properties:
          phi:
            $ref: "angle-1.0.0"
          theta:
            $ref: "angle-1.0.0"
          r:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"

      # CylindricalRepresentation
      - type: object
        properties:
          rho:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          phi:
            $ref: "angle-1.0.0"
          z:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"

      # CartesianDifferential
      - type: object
        properties:
          d_x:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          d_y:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          d_z:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"

      # SphericalDifferential
      - type: object
        properties:
          d_lon:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          d_lat:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          d_distance:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"

      # UnitSphericalCosLatDifferential
      - type: object
        properties:
          d_lon_coslat:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          d_lat:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"

      # UnitSphericalDifferential
      - type: object
        properties:
          d_lon:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          d_lat:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"

      # SphericalCosLatDifferential
      - type: object
        properties:
          d_lon_coslat:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          d_lat:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          d_distance:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
      # SphericalDifferential
      - type: object
        properties:
          d_lon:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          d_lat:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          d_distance:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"

      # RadialDifferential
      - type: object
        properties:
          d_phi:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          d_theta:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          d_r:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"

      # PhysicsSphericalDifferential
      - type: object
        properties:
          d_phi:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          d_theta:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          d_r:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"

      # RadialDifferential
      - type: object
        properties:
          d_distance:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"

      # CylindricalDifferential
      - type: object
        properties:
          d_rho:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          d_phi:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
          d_z:
            $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"

required: [type, components]
...
```

coordinates/skycoord-1.0.0

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://astropy.org/schemas/astropy/coordinates/skycoord-1.0.0"
tag: "tag:astropy.org:astropy/coordinates/skycoord-1.0.0"

title: |
  Represents a SkyCoord object from astropy

allOf:
  - type: object
    properties:
      frame:
        description: |
          A string describing the kind of frame that is represented
by this
          SkyCoord object. This value is used when reconstructing
SkyCoord.
        type: string
required: [frame]
additionalProperties: true
...
```

FITS

The following schemas are associated with `astropy` types from the FITS File Handling (astropy.io.fits) submodule:

fits/fits-1.0.0

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://astropy.org/schemas/astropy/fits/fits-1.0.0"
title: >
  A FITS file inside of an ASDF file.
description: |
  This schema is useful for distributing ASDF files that can
  automatically be converted to FITS files by specifying the exact
  content of the resulting FITS file.

  Not all kinds of data in FITS are directly representable in ASDF.
  For example, applying an offset and scale to the data using the
  `BZERO` and `BSCALE` keywords.  In these cases, it will not be
  possible to store the data in the native format from FITS and also
  be accessible in its proper form in the ASDF file.
  Only image and binary table extensions are supported.

examples:
  -
    - A simple FITS file with a primary header and two extensions
    - |
        !<tag:astropy.org:astropy/fits/fits-1.0.0>
            - header:
              - [SIMPLE, true, conforms to FITS standard]
              - [BITPIX, 8, array data type]
              - [NAXIS, 0, number of array dimensions]
              - [EXTEND, true]
              - []
              - ['', Top Level MIRI Metadata]
              - []
              - [DATE, '2013-08-30T10:49:55.070373', The date this
file was created (UTC)]
              - [FILENAME, MiriDarkReferenceModel_test.fits, The name
of the file]
              - [TELESCOP, JWST, The telescope used to acquire the
data]
              - []
              - ['', Information about the observation]
              - []
              - [DATE-OBS, '2013-08-30T10:49:55.000000', The date the
observation was made (UTC)]
            - data: !core/ndarray-1.0.0
                datatype: float32
                shape: [2, 3, 3, 4]
                source: 0
                byteorder: big
              header:
              - [XTENSION, IMAGE, Image extension]
              - [BITPIX, -32, array data type]
              - [NAXIS, 4, number of array dimensions]
              - [NAXIS1, 4]
              - [NAXIS2, 3]
              - [NAXIS3, 3]
              - [NAXIS4, 2]
              - [PCOUNT, 0, number of parameters]
              - [GCOUNT, 1, number of groups]
              - [EXTNAME, SCI, extension name]
              - [BUNIT, DN, Units of the data array]
            - data: !core/ndarray-1.0.0
                datatype: float32
                shape: [2, 3, 3, 4]
                source: 1
                byteorder: big
              header:
```

```
              - [XTENSION, IMAGE, Image extension]
              - [BITPIX, -32, array data type]
              - [NAXIS, 4, number of array dimensions]
              - [NAXIS1, 4]
              - [NAXIS2, 3]
              - [NAXIS3, 3]
              - [NAXIS4, 2]
              - [PCOUNT, 0, number of parameters]
              - [GCOUNT, 1, number of groups]
              - [EXTNAME, ERR, extension name]
              - [BUNIT, DN, Units of the error array]

allOf:
  - tag: "tag:astropy.org:astropy/fits/fits-1.0.0"
  - type: array
    items:
      type: object
      properties:
        data:
          description: "The data part of the HDU."
          anyOf:
            - $ref: "tag:stsci.edu:asdf/core/ndarray-1.0.0"
            - $ref: "../table/table-1.0.0"
            # Retain backwards compatibility with table defined by
ASDF Standard
            - $ref: "tag:stsci.edu:asdf/core/table-1.0.0"
            - type: "null"
          default: null
```

Table

The following schemas are associated with `astropy` types from the Data Tables (astropy.table) submodule:

table/table-1.0.0

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://astropy.org/schemas/astropy/table/table-1.0.0"
tag: "tag:astropy.org:astropy/table/table-1.0.0"

title: >
  A table.

description: |
  A table is represented as a list of columns, where each entry is a
  [column](ref:http://stsci.edu/schemas/asdf/core/column-1.0.0)
  object, containing the data and some additional information.
  The data itself may be stored inline as text, or in binary in
either
  row- or column-major order by use of the `strides` property on the
  individual column arrays.

  Each column in the table must have the same first (slowest moving)
  dimension.

examples:
  -
    - A table stored in column-major order, with each column in a
separate block
    - |
        !<tag:astropy.org:astropy/table/table-1.0.0>
          columns:
          - !core/column-1.0.0
            data: !core/ndarray-1.0.0
              source: 0
              datatype: float64
              byteorder: little
              shape: [3]
            description: RA
            meta: {foo: bar}
            name: a
            unit: !unit/unit-1.0.0 deg
          - !core/column-1.0.0
            data: !core/ndarray-1.0.0
              source: 1
              datatype: float64
              byteorder: little
              shape: [3]
            description: DEC
            name: b
          - !core/column-1.0.0
            data: !core/ndarray-1.0.0
              source: 2
              datatype: [ascii, 1]
              byteorder: big
              shape: [3]
            description: The target name
            name: c
          colnames: [a, b, c]

  -
    - A table stored in row-major order, all stored in the same block
    - |
        !<tag:astropy.org:astropy/table/table-1.0.0>
          columns:
```

```
          - !core/column-1.0.0
            data: !core/ndarray-1.0.0
              source: 0
              datatype: float64
              byteorder: little
              shape: [3]
              strides: [13]
            description: RA
            meta: {foo: bar}
            name: a
            unit: !unit/unit-1.0.0 deg
          - !core/column-1.0.0
            data: !core/ndarray-1.0.0
              source: 0
              datatype: float64
              byteorder: little
              shape: [3]
              offset: 4
              strides: [13]
            description: DEC
            name: b
          - !core/column-1.0.0
            data: !core/ndarray-1.0.0
              source: 0
              datatype: [ascii, 1]
              byteorder: big
              shape: [3]
              offset: 12
              strides: [13]
            description: The target name
            name: c
          colnames: [a, b, c]

type: object
properties:
  columns:
    description: |
      A list of columns in the table.
    type: array
    items:
      anyOf:
        - $ref: "tag:stsci.edu:asdf/core/column-1.0.0"
        - $ref: "tag:stsci.edu:asdf/core/ndarray-1.0.0"
        - $ref: "tag:stsci.edu:asdf/time/time-1.1.0"
        - $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
        - $ref: "../coordinates/skycoord-1.0.0"
        - $ref: "../coordinates/earthlocation-1.0.0"
        - $ref: "../time/timedelta-1.0.0"
  colnames:
    description: |
      A list containing the names of the columns in the table (in
order).
    type: array
    items:
      - type: string

  qtable:
    description: |
      A flag indicating whether or not the serialized type was a
QTable
    type: boolean
    default: False

  meta:
    description: |
      Additional free-form metadata about the table.
    type: object
    default: {}

additionalProperties: false
required: [columns, colnames]
```

Time

The following schemas are associated with `astropy` types from the Time and Dates (astropy.time) submodule:

time/timedelta-1.0.0

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/asdf/asdf-schema-1.0.0"
id: "http://astropy.org/schemas/astropy/time/time-1.1.0"
title: Represents an instance of TimeDelta from astropy
description: |
  Represents the time difference between two times.

type: object
properties:
    jd1:
      anyOf:
        - type: number
        - $ref: "tag:stsci.edu:asdf/core/ndarray-1.0.0"
      description: |
        Value representing first 64 bits of precision
    jd2:
      anyOf:
        - type: number
        - $ref: "tag:stsci.edu:asdf/core/ndarray-1.0.0"
      description: |
        Value representing second 64 bits of precision
    format:
      type: string
      description: |
        Format of time value representation.
    scale:
      type: string
      description: |
        Time scale of input value(s).
      enum: [tdb, tt, ut1, tcg, tcb, tai, local]
required: [jd1, jd2, format]
additionalProperties: False
...
```

Units

The following schemas are associated with `astropy` types from the Units and Quantities (astropy.units) submodule:

units/equivalency-1.0.0

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
id: "http://astropy.org/schemas/astropy/units/equivalency-1.0.0"
tag: "tag:astropy.org:astropy/units/equivalency-1.0.0"

title: |
  Represents unit equivalency.

description: |
  Supports serialization of equivalencies between units
  in certain contexts

definitions:
  equivalency:
    type: object
    properties:
      name:
        type: string
      kwargs_names:
        type: array
        items:
          type: string
      kwargs_values:
        type: array
        items:
          anyOf:
            - $ref: "tag:stsci.edu:asdf/unit/quantity-1.1.0"
            - type: number
            - type: "null"

type: array
items:
  $ref: "#/definitions/equivalency"
...
```

SAMP (Simple Application Messaging Protocol) (`astropy.samp`)

`astropy.samp` is a Python implementation of the SAMP messaging system.

Simple Application Messaging Protocol (SAMP) is an inter-process communication system that allows different client programs, usually running on the same computer, to communicate with each other by exchanging short messages that may reference external data files. The protocol has been developed within the International Virtual Observatory Alliance (IVOA) and is understood by many desktop astronomy tools, including TOPCAT, SAO DS9, and Aladin.

So by using the classes in **`astropy.samp`**, Python code can interact with other running desktop clients, for instance displaying a named FITS file in DS9, prompting Aladin to recenter on a given sky position, or receiving a message identifying the row when a user highlights a plotted point in TOPCAT.

The way the protocol works is that a SAMP “Hub” process must be running on the local host, and then various client programs can connect to it. Once connected, these clients can send messages to each other via the hub. The details are described in the SAMP standard.

`astropy.samp` provides classes both to set up such a hub process, and to help implement a client that can send and receive messages. It also provides a stand-alone program `samp_hub` which can run a persistent hub in its own process. Note that setting up the hub from Python is not always necessary, since various other SAMP-aware applications may start up a hub independently; in most cases, only one running hub is used during a SAMP session.

The following classes are available in **`astropy.samp`**:

- **`SAMPHubServer`**, which is used to instantiate a hub server that clients can then connect to.
- **`SAMPHubProxy`**, which is used to connect to an existing hub (including hubs

started from other applications such as TOPCAT).
- `SAMPClient`, which is used to create a SAMP client.
- `SAMPIntegratedClient`, which is the same as `SAMPClient` except that it has a self-contained `SAMPHubProxy` to provide a simpler user interface.

`astropy.samp` is a full implementation of SAMP V1.3. As well as the Standard Profile, it supports the Web Profile, which means that it can be used to also communicate with web SAMP clients; see the sampjs library examples for more details.

Using `astropy.samp`

Starting and Stopping a SAMP Hub Server

There are several ways you can start up a SAMP hub:

Using an Existing Hub

You can start up another application that includes a hub, such as TOPCAT, SAO Ds9, or Aladin Desktop.

Using the Command-Line Hub Utility

You can make use of the `samp_hub` command-line utility, which is included in `astropy`:

```
$ samp_hub
```

To get more help on available options for `samp_hub`:

```
$ samp_hub -h
```

To stop the server, press control-C.

Starting a Hub Programmatically (Advanced)

You can start up a hub by creating a `SAMPHubServer` instance and starting it, either from the interactive Python prompt, or from a Python script:

```
>>> from astropy.samp import SAMPHubServer
>>> hub = SAMPHubServer()
>>> hub.start()
```

You can then stop the hub by calling:

```
>>> hub.stop()
```

However, this method is generally not recommended for average users because it does not work correctly when web SAMP clients try to connect. Instead, this should be reserved for developers who want to embed a SAMP hub in a GUI, for example. For more information, see Embedding a SAMP Hub in a GUI.

Sending and Receiving Tables and Images over SAMP

In the following examples, we make use of:

- TOPCAT, which is a tool to explore tabular data.
- SAO DS9, which is an image visualization tool that can overplot catalogs.
- Aladin Desktop, which is another tool that can visualize images and catalogs.

TOPCAT and Aladin will run a SAMP Hub if none is found, so for the following examples you can either start up one of these applications first, or you can start up the `astropy.samp` hub. You can start this using the following command:

```
$ samp_hub
```

Sending a Table to TOPCAT and DS9

The easiest way to send a VO table to TOPCAT is to make use of the `SAMPIntegratedClient` class. Once TOPCAT is open, first instantiate a `SAMPIntegratedClient` instance and then connect to the hub:

```
>>> from astropy.samp import SAMPIntegratedClient
>>> client = SAMPIntegratedClient()
>>> client.connect()
```

Next, we have to set up a dictionary that contains details about the table to send. This should include `url`, which is the URL to the file, and `name`, which is a human-readable name for the table. The URL can be a local URL (starting with `file:///`):

```
>>> params = {}
>>> params["url"] = 'file:///Users/tom/Desktop
/aj285677t3_votable.xml'
>>> params["name"] = "Robitaille et al. (2008), Table 3"
```

Note

To construct a local URL, you can also make use of `urlparse` as follows:

```
>>> import urlparse
>>> params["url"] = urlparse.urljoin('file:',
os.path.abspath("aj285677t3_votable.xml"))
```

Now we can set up the message itself. This includes the type of message (here we use `table.load.votable`, which indicates that a VO table should be loaded and the details of the table that we set above):

```
>>> message = {}
>>> message["samp.mtype"] = "table.load.votable"
>>> message["samp.params"] = params
```

Finally, we can broadcast this to all clients that are listening for `table.load.votable` messages using **`notify_all()`**:

```
>>> client.notify_all(message)
```

The above message will actually be broadcast to all applications connected via SAMP. For example, if we open SAO DS9 in addition to TOPCAT, and we run the above command, both applications will load the table. We can use the **`get_registered_clients()`** method to find all of the clients connected to the hub:

```
>>> client.get_registered_clients()
['hub', 'c1', 'c2']
```

These IDs do not mean much, but we can find out more using:

```
>>> client.get_metadata('c1')
{'author.affiliation': 'Astrophysics Group, Bristol University',
 'author.email': 'm.b.taylor@bristol.ac.uk',
 'author.name': 'Mark Taylor',
 'home.page': 'http://www.starlink.ac.uk/topcat/',
 'samp.description.text': 'Tool for OPerations on Catalogues And
Tables',
 'samp.documentation.url': 'http://127.0.0.1:2525/doc/sun253
/index.html',
 'samp.icon.url': 'http://127.0.0.1:2525/doc/images/tc_sok.gif',
 'samp.name': 'topcat',
 'topcat.version': '4.0-1'}
```

We can see that `c1` is the TOPCAT client. We can now resend the data, but this time only to TOPCAT, using the **`notify()`** method:

```
>>> client.notify('c1', message)
```

Once finished, we should make sure we disconnect from the hub:

```
>>> client.disconnect()
```

Receiving a Table from TOPCAT

To receive a table from TOPCAT, we have to set up a client that listens for messages from the hub. As before, we instantiate a **`SAMPIntegratedClient`** instance and connect to the hub:

```
>>> from astropy.samp import SAMPIntegratedClient
>>> client = SAMPIntegratedClient()
>>> client.connect()
```

We now set up a receiver class which will handle any received messages. We need to take care to write handlers for both notifications and calls (the difference between the two being that calls expect a reply):

```
>>> class Receiver(object):
...     def __init__(self, client):
...         self.client = client
...         self.received = False
...     def receive_call(self, private_key, sender_id, msg_id, mtype,
params, extra):
...         self.params = params
...         self.received = True
...         self.client.reply(msg_id, {"samp.status": "samp.ok",
"samp.result": {}})
...     def receive_notification(self, private_key, sender_id, mtype,
params, extra):
...         self.params = params
...         self.received = True
```

And we instantiate it:

```
>>> r = Receiver(client)
```

We can now use the **`bind_receive_call()`** and **`bind_receive_notification()`** methods to tell our receiver to listen to all `table.load.votable` messages:

```
>>> client.bind_receive_call("table.load.votable", r.receive_call)
>>> client.bind_receive_notification("table.load.votable",
r.receive_notification)
```

We can now check that the message has not been received yet:

```
>>> r.received
False
```

We can now broadcast the table from TOPCAT. After a few seconds, we can check again if the message has been received:

```
>>> r.received
True
```

Success! The table URL should now be available in `r.params['url']`, so we can do:

```
>>> from astropy.table import Table
>>> t = Table.read(r.params['url'])
Downloading http://127.0.0.1:2525/dynamic/4/t12.vot [Done]
>>> t
```

```
           col1                col2     col3     col4     col5     col6
col7  col8 col9 col10
------------------------- -------- ------- -------- -------- -----
---- ----- ---- -----
SSTGLMC G000.0046+01.1431   0.0046  1.1432 265.2992 -28.3321  6.67
5.04  6.89 5.22     N
SSTGLMC G000.0106-00.7315   0.0106 -0.7314 267.1274 -29.3063  7.18
6.07   nan 5.17     Y
SSTGLMC G000.0110-01.0237   0.0110 -1.0236 267.4151 -29.4564  8.32
6.30  8.34 6.32     N
...
```

As before, we should remember to disconnect from the hub once we are done:

```
>>> client.disconnect()
```

Example

The following is a full example of a script that can be used to receive and read a table. It includes a loop that waits until the message is received, and reads the table once it has:

```
import time

from astropy.samp import SAMPIntegratedClient
from astropy.table import Table

 # Instantiate the client and connect to the hub
client=SAMPIntegratedClient()
client.connect()

# Set up a receiver class
class Receiver(object):
    def __init__(self, client):
        self.client = client
        self.received = False
    def receive_call(self, private_key, sender_id, msg_id, mtype,
params, extra):
        self.params = params
        self.received = True
        self.client.reply(msg_id, {"samp.status": "samp.ok",
"samp.result": {}})
    def receive_notification(self, private_key, sender_id, mtype,
params, extra):
        self.params = params
        self.received = True

# Instantiate the receiver
r = Receiver(client)

# Listen for any instructions to load a table
client.bind_receive_call("table.load.votable", r.receive_call)
client.bind_receive_notification("table.load.votable",
r.receive_notification)

# We now run the loop to wait for the message in a try/finally block
so that if
# the program is interrupted e.g. by control-C, the client terminates
# gracefully.

try:

    # We test every 0.1s to see if the hub has sent a message
    while True:
        time.sleep(0.1)
        if r.received:
            t = Table.read(r.params['url'])
            break

finally:

    client.disconnect()

# Print out table
print t
```

Sending an Image to DS9 and Aladin

As for tables, the most convenient way to send a FITS image over SAMP is to make use of the `SAMPIntegratedClient` class. Once Aladin or DS9 are open, first instantiate a `SAMPIntegratedClient` instance and then connect to the hub as before:

```
>>> from astropy.samp import SAMPIntegratedClient
>>> client = SAMPIntegratedClient()
>>> client.connect()
```

Next, we have to set up a dictionary that contains details about the image to send. This should include `url`, which is the URL to the file, and `name`, which is a human-readable name for the table. The URL can be a local URL (starting with `file:///`):

```
>>> params = {}
>>> params["url"] = 'file:///Users/tom/Desktop/MSX_E.fits'
>>> params["name"] = "MSX Band E Image of the Galactic Center"
```

See Sending a Table to TOPCAT and DS9 for an example of a recommended way to construct local URLs. Now we can set up the message itself. This includes the type of message (here we use `image.load.fits` which indicates that a FITS image should be loaded, and the details of the table that we set above):

```
>>> message = {}
>>> message["samp.mtype"] = "image.load.fits"
>>> message["samp.params"] = params
```

Finally, we can broadcast this to all clients that are listening for `table.load.votable` messages:

```
>>> client.notify_all(message)
```

As for Sending a Table to TOPCAT and DS9, the **`notify_all()`** method will broadcast the image to all listening clients, and for tables it is possible to instead use the **`notify()`** method to send it to a specific client.

Once finished, we should make sure we disconnect from the hub:

```
>>> client.disconnect()
```

Receiving a Table from DS9 or Aladin

Receiving images over SAMP is identical to Receiving a Table from TOPCAT, with the exception that the message type should be `image.load.fits` instead of `table.load.votable`. Once the URL has been received, the FITS image can be opened with:

```
>>> from astropy.io import fits
>>> fits.open(r.params['url'])
```

Communication between Integrated Clients Objects

As shown in Sending and Receiving Tables and Images over SAMP, the **SAMPIntegratedClient** class can be used to communicate with other SAMP-enabled tools such as TOPCAT, SAO DS9, or Aladin Desktop.

In this section, we look at how we can set up two **SAMPIntegratedClient** instances and communicate between them.

First, start up a SAMP hub as described in Starting and Stopping a SAMP Hub Server.

Next, we create two clients and connect them to the hub:

```
>>> from astropy import samp
>>> client1 = samp.SAMPIntegratedClient(name="Client 1",
description="Test Client 1",
...                                     metadata =
{"client1.version":"0.01"})
>>> client2 = samp.SAMPIntegratedClient(name="Client 2",
description="Test Client 2",
...                                     metadata =
{"client2.version":"0.25"})
>>> client1.connect()
>>> client2.connect()
```

We now define functions to call when receiving a notification, call or response:

```
>>> def test_receive_notification(private_key, sender_id, mtype,
params, extra):
...     print("Notification:", private_key, sender_id, mtype, params,
extra)

>>> def test_receive_call(private_key, sender_id, msg_id, mtype,
params, extra):
...     print("Call:", private_key, sender_id, msg_id, mtype, params,
extra)
...     client1.ereply(msg_id, samp.SAMP_STATUS_OK, result = {"txt":
"printed"})

>>> def test_receive_response(private_key, sender_id, msg_id,
response):
...     print("Response:", private_key, sender_id, msg_id, response)
```

We subscribe client 1 to `"samp.app.*"` and bind it to the related functions:

```
>>> client1.bind_receive_notification("samp.app.*",
test_receive_notification)
>>> client1.bind_receive_call("samp.app.*", test_receive_call)
```

We now bind message tags received by client 2 to suitable functions:

```
>>> client2.bind_receive_response("my-dummy-print",
test_receive_response)
>>> client2.bind_receive_response("my-dummy-print-specific",
test_receive_response)
```

We are now ready to test out the clients and callback functions. Client 2 notifies all clients using the “samp.app.echo” message type via the hub:

```
>>> client2.enotify_all("samp.app.echo", txt="Hello world!")
['cli#2']
Notification: 0d7f4500225981c104a197c7666a8e4e cli#2 samp.app.echo
{'txt':
'Hello world!'} {'host': 'antigone.lambrate.inaf.it', 'user':
'unknown'}
```

We can also find a dictionary that specifies which clients would currently receive `samp.app.echo` messages:

```
>>> print(client2.get_subscribed_clients("samp.app.echo"))
{'cli#2': {}}
```

Client 2 calls all clients with the `"samp.app.echo"` message type using `"my-dummy-print"` as a message-tag:

```
>>> print(client2.call_all("my-dummy-print",
...                        {"samp.mtype": "samp.app.echo",
...                         "samp.params": {"txt": "Hello world!"}}))
{'cli#1': 'msg#1;;cli#hub;;cli#2;;my-dummy-print'}
Call: 8c8eb53178cb95e168ab17ec4eac2353 cli#2
msg#1;;cli#hub;;cli#2;;my-dummy-print samp.app.echo {'txt': 'Hello
world!'}
{'host': 'antigone.lambrate.inaf.it', 'user': 'unknown'}
Response: d0a28636321948ccff45edaf40888c54 cli#1 my-dummy-print
{'samp.status': 'samp.ok', 'samp.result': {'txt': 'printed'}}
```

Client 2 then calls client 1 using the `"samp.app.echo"` message type, tagging the message as `"my-dummy-print-specific"`:

```
>>> try:
...     print(client2.call(client1.get_public_id(),
...                        "my-dummy-print-specific",
...                        {"samp.mtype": "samp.app.echo",
...                         "samp.params": {"txt": "Hello client
1!"}}))
... except samp.SAMPProxyError as e:
...     print("Error ({0}): {1}".format(e.faultCode, e.faultString))
msg#2;;cli#hub;;cli#2;;my-dummy-print-specific
Call: 8c8eb53178cb95e168ab17ec4eac2353 cli#2
msg#2;;cli#hub;;cli#2;;my-dummy-print-specific samp.app.echo {'txt':
'Hello
Cli 1!'} {'host': 'antigone.lambrate.inaf.it', 'user': 'unknown'}
Response: d0a28636321948ccff45edaf40888c54 cli#1 my-dummy-print-
specific
{'samp.status': 'samp.ok', 'samp.result': {'txt': 'printed'}}
```

We can now define a function called to test synchronous calls:

```
>>> def test_receive_sync_call(private_key, sender_id, msg_id, mtype,
params, extra):
...     import time
...     print("SYNC Call:", sender_id, msg_id, mtype, params, extra)
...     time.sleep(2)
...     client1.reply(msg_id, {"samp.status": samp.SAMP_STATUS_OK,
...                            "samp.result": {"txt": "printed
sync"}})
```

We now bind the `samp.test` message type to `test_receive_sync_call`:

```
>>> client1.bind_receive_call("samp.test", test_receive_sync_call)
>>> try:
...     # Sync call
...     print(client2.call_and_wait(client1.get_public_id(),
...                                 {"samp.mtype": "samp.test",
...                                  "samp.params": {"txt": "Hello
SYNCRO client 1!"}},
...                                  "10"))
... except samp.SAMPProxyError as e:
...     # If timeout expires than a SAMPProxyError is returned
...     print("Error ({0}): {1}".format(e.faultCode, e.faultString))
SYNC Call: cli#2 msg#3;;cli#hub;;cli#2;;sampy::sync::call samp.test
{'txt':
'Hello SYNCRO Cli 1!'} {'host': 'antigone.lambrate.inaf.it', 'user':
'unknown'}
{'samp.status': 'samp.ok', 'samp.result': {'txt': 'printed sync'}}
```

Finally, we disconnect the clients from the hub at the end:

```
>>> client1.disconnect()
>>> client2.disconnect()
```

Embedding a SAMP Hub in a GUI

Overview

If you wish to embed a SAMP hub in your Python Graphical User Interface (GUI) tool, you will need to start the hub programmatically using:

```
from astropy.samp import SAMPHubServer
hub = SAMPHubServer()
hub.start()
```

This launches the hub in a thread and is non-blocking. If you are not interested in connections from web SAMP clients, then you can use:

```
from astropy.samp import SAMPHubServer
hub = SAMPHubServer(web_profile=False)
hub.start()
```

This should be all you need to do. However, if you want to keep the Web Profile active, there is an additional consideration: when a web SAMP client connects, you will need to ask the user whether they accept the connection (for security reasons). By default, the confirmation message is a text-based message in the terminal, but if you have a GUI tool, you will likely want to open a GUI dialog instead.

To do this, you will need to define a class that handles the dialog, and then pass an **instance** of the class to `SAMPHubServer` (not the class itself). This class should inherit from `astropy.samp.WebProfileDialog` and add the following:

1. A GUI timer callback that periodically calls `WebProfileDialog.handle_queue` (available as `self.handle_queue`).
2. A `show_dialog` method to display a consent dialog. It should take the following arguments:
    - `samp_name`: The name of the application making the request.

- `details` : A dictionary of details about the client making the request. The only key in this dictionary required by the SAMP standard is `samp.name` which gives the name of the client making the request.
- `client` : A hostname, port pair containing the client address.
- `origin` : A string containing the origin of the request.

3. Based on the user response, the `show_dialog` should call `WebProfileDialog.consent` or `WebProfileDialog.reject`. This may, in some cases, be the result of another GUI callback.

Example of embedding a SAMP hub in a Tk application

The following code is a full example of a Tk application that watches for web SAMP connections and opens the appropriate dialog:

```
import tkinter as tk
import tkinter.messagebox as tkMessageBox

from astropy.samp import SAMPHubServer
from astropy.samp.hub import WebProfileDialog

MESSAGE = """
A Web application which declares to be

Name: {name}
Origin: {origin}

is requesting to be registered with the SAMP Hub.  Pay attention
that if you permit its registration, such application will acquire
all current user privileges, like file read/write.

Do you give your consent?
"""


class TkWebProfileDialog(WebProfileDialog):
    def __init__(self, root):
        self.root = root
        self.wait_for_dialog()

    def wait_for_dialog(self):
        self.handle_queue()
        self.root.after(100, self.wait_for_dialog)

    def show_dialog(self, samp_name, details, client, origin):
        text = MESSAGE.format(name=samp_name, origin=origin)

        response = tkMessageBox.askyesno(
            'SAMP Hub', text,
            default=tkMessageBox.NO)

        if response:
            self.consent()
        else:
            self.reject()

# Start up Tk application
root = tk.Tk()
tk.Label(root, text="Example SAMP Tk application",
         font=("Helvetica", 36), justify=tk.CENTER).pack(pady=200)
root.geometry("500x500")
root.update()

# Start up SAMP hub
h = SAMPHubServer(web_profile_dialog=TkWebProfileDialog(root))
h.start()

try:
    # Main GUI loop
    root.mainloop()
except KeyboardInterrupt:
    pass

h.stop()
```

If you run the above script, a window will open that says “Example SAMP Tk application.” If you then go to the following page, for example:

http://astrojs.github.io/sampjs/examples/pinger.html

and click on the Ping button, you will see the dialog open in the Tk application. Once you click on “CONFIRM,” future “Ping” calls will no longer bring up the dialog.

Reference/API

astropy.samp Package

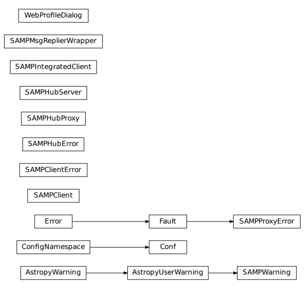
This subpackage provides classes to communicate with other applications via the Simple Application Messaging Protocal (SAMP).

Before integration into Astropy it was known as SAMPy, and was developed by Luigi Paioro (INAF - Istituto Nazionale di Astrofisica).

Classes

`Conf`()	Configuration parameters for `astropy.samp`.
`SAMPClient`(hub[, name, description, …])	Utility class which provides facilities to create and manage a SAMP compliant XML-RPC server that acts as SAMP callable client application.
`SAMPClientError`	SAMP Client exceptions.
`SAMPHubError`	SAMP Hub exception.
`SAMPHubProxy`()	Proxy class to simplify the client interaction with a SAMP hub (via the standard profile).
`SAMPHubServer`([secret, addr, port, …])	SAMP Hub Server.
`SAMPIntegratedClient`([name, description, …])	A Simple SAMP client.
`SAMPMsgReplierWrapper`(cli)	Function decorator that allows to automatically grab errors and returned maps (if any) from a function bound to a SAMP call (or notify).
`SAMPProxyError`(faultCode, faultString, **extra)	SAMP Proxy Hub exception
`SAMPWarning`	SAMP-specific Astropy warning class
`WebProfileDialog`()	A base class to make writing Web Profile GUI consent dialogs easier.

Class Inheritance Diagram



Acknowledgments

This code is adapted from the SAMPy package written by Luigi Paioro, who has granted the Astropy Project permission to use the code under a BSD license.

Computations and utilities

Cosmological Calculations (`astropy.cosmology`)

Introduction

The **astropy.cosmology** sub-package contains classes for representing cosmologies and utility functions for calculating commonly used quantities that depend on a cosmological model. This includes distances, ages, and lookback times corresponding to a measured redshift or the transverse separation corresponding to a measured angular separation.

Getting Started

Cosmological quantities are calculated using methods of a **Cosmology** object.

Examples

To calculate the Hubble constant at z=0 (i.e., `H0`) and the number of transverse proper kiloparsecs (kpc) corresponding to an arcminute at z=3:

```
>>> from astropy.cosmology import WMAP9 as cosmo
>>> cosmo.H(0)
<Quantity 69.32 km / (Mpc s)>
```

```
>>> cosmo.kpc_proper_per_arcmin(3)
<Quantity 472.97709620405266 kpc / arcmin>
```

Here WMAP9 is a built-in object describing a cosmology with the parameters from the nine-year WMAP results. Several other built-in cosmologies are also available (see Built-in Cosmologies). The available methods of the cosmology object are listed in the methods summary for the **FLRW** class. If you are using IPython you can also use tab completion to print a list of the available methods. To do this, after importing the cosmology as in the above example, type `cosmo.` at the IPython prompt and then press the tab key.

All of these methods also accept an arbitrarily-shaped array of redshifts as input:

```
>>> from astropy.cosmology import WMAP9 as cosmo
>>> cosmo.comoving_distance([0.5, 1.0, 1.5])
<Quantity [1916.06941724, 3363.07062107, 4451.7475201 ] Mpc>
```

You can create your own FLRW-like cosmology using one of the cosmology classes:

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> cosmo = FlatLambdaCDM(H0=70, Om0=0.3, Tcmb0=2.725)
>>> cosmo
FlatLambdaCDM(H0=70 km / (Mpc s), Om0=0.3, Tcmb0=2.725 K,
              Neff=3.04, m_nu=[0. 0. 0.] eV, Ob0=None)
```

Note the presence of additional cosmological parameters (e.g., `Neff`, the number of effective neutrino species) with default values; these can also be specified explicitly in the call to the constructor.

The cosmology sub-package makes use of **`units`**, so in many cases returns values with units attached. Consult the documentation for that sub-package for more details, but briefly here we will show how to access the floating point or array values:

```
>>> from astropy.cosmology import WMAP9 as cosmo
>>> H0 = cosmo.H(0)
>>> H0.value, H0.unit
(69.32, Unit("km / (Mpc s)"))
```

Using astropy.cosmology

Most of the functionality is enabled by the **`FLRW`** object. This represents a homogeneous and isotropic cosmology (characterized by the Friedmann-Lemaitre-Robertson-Walker metric, named after the people who solved Einstein's field equation for this special case). However, you cannot work with this class directly, as you must specify a dark energy model by using one of its subclasses instead, such as **`FlatLambdaCDM`**.

Examples

You can create a new **`FlatLambdaCDM`** object with arguments giving the Hubble parameter and Omega matter (both at z=0):

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> cosmo = FlatLambdaCDM(H0=70, Om0=0.3)
>>> cosmo
FlatLambdaCDM(H0=70 km / (Mpc s), Om0=0.3, Tcmb0=0 K,
              Neff=3.04, m_nu=None, Ob0=None)
```

This can also be done more explicitly using units, which is recommended:

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> import astropy.units as u
>>> cosmo = FlatLambdaCDM(H0=70 * u.km / u.s / u.Mpc, Tcmb0=2.725 *
u.K, Om0=0.3)
```

However, most of the parameters that accept units (`H0`, `Tcmb0`) have default units, so unit quantities do not have to be used (with the exception of neutrino masses, where you must supply a unit if you want massive neutrinos).

The predefined cosmologies described in the Getting Started section are instances of **`FlatLambdaCDM`**, and have the same methods. So we can find the luminosity distance to redshift 4 by:

```
>>> cosmo.luminosity_distance(4)
<Quantity 35842.353618623194 Mpc>
```

Or the age of the universe at z = 0:

```
>>> cosmo.age(0)
<Quantity 13.461701658024014 Gyr>
```

They also accept arrays of redshifts:

```
>>> cosmo.age([0.5, 1, 1.5]).value
array([8.42128013, 5.74698021, 4.19645373])
```

See the **`FLRW`** and **`FlatLambdaCDM`** object docstring for all of the methods and attributes available.

In addition to flat universes, non-flat varieties are supported, such as **`LambdaCDM`**. A variety of standard cosmologies with the parameters already defined are also available (see Built-in Cosmologies):

```
>>> from astropy.cosmology import WMAP7   # WMAP 7-year cosmology
>>> WMAP7.critical_density(0)  # critical density at z = 0
<Quantity 9.31000324385361e-30 g / cm3>
```

You can see how the density parameters evolve with redshift as well:

```
>>> from astropy.cosmology import WMAP7   # WMAP 7-year cosmology
>>> WMAP7.Om([0, 1.0, 2.0]), WMAP7.Ode([0., 1.0, 2.0])
(array([0.272     , 0.74898522, 0.90905234]),
 array([0.72791572, 0.2505506 , 0.0901026 ]))
```

Note that these do not quite add up to one, even though WMAP7 assumes a flat universe, because photons and neutrinos are included. Also note that the density parameters are unitless and so are not **`Quantity`** objects.

It is possible to specify the baryonic matter density at redshift zero at class instantiation by passing the keyword argument `Ob0`:

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> cosmo = FlatLambdaCDM(H0=70, Om0=0.3, Ob0=0.05)
>>> cosmo
FlatLambdaCDM(H0=70 km / (Mpc s), Om0=0.3, Tcmb0=0 K,
               Neff=3.04, m_nu=None, Ob0=0.05)
```

In this case the dark matter-only density at redshift 0 is available as class attribute `Odm0` and the redshift evolution of dark and baryonic matter densities can be computed using the methods `Odm` and `Ob`, respectively. If `Ob0` is not specified at class instantiation, it defaults to `None` and any method relying on it being specified will raise a `ValueError`:

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> cosmo = FlatLambdaCDM(H0=70, Om0=0.3)
>>> cosmo.Odm(1)
Traceback (most recent call last):
...
ValueError: Baryonic density not set for this cosmology, unclear
meaning of dark matter density
```

Cosmological instances have an optional `name` attribute which can be used to describe the cosmology:

```
>>> from astropy.cosmology import FlatwCDM
>>> cosmo = FlatwCDM(name='SNLS3+WMAP7', H0=71.58, Om0=0.262,
w0=-1.016)
>>> cosmo
FlatwCDM(name="SNLS3+WMAP7", H0=71.6 km / (Mpc s), Om0=0.262,
         w0=-1.02, Tcmb0=0 K, Neff=3.04, m_nu=None, Ob0=None)
```

This is also an example with a different model for dark energy: a flat universe with a constant dark energy equation of state, but not necessarily a cosmological constant. A variety of additional dark energy models are also supported (see Specifying a dark energy model).

An important point is that the cosmological parameters of each instance are immutable — that is, if you want to change, say, `Om`, you need to make a new instance of the class. To make this more convenient, a `clone` operation is provided, which allows you to make a copy with specified values changed. Note that you cannot change the type of cosmology with this operation (e.g., flat to non-flat).

To make a copy of a cosmological instance using the `clone` operation:

```
>>> from astropy.cosmology import WMAP9
>>> newcosmo = WMAP9.clone(name='WMAP9 modified', Om0=0.3141)
>>> WMAP9.H0, newcosmo.H0  # some values unchanged
(<Quantity 69.32 km / (Mpc s)>, <Quantity 69.32 km / (Mpc s)>)
>>> WMAP9.Om0, newcosmo.Om0  # some changed
(0.2865, 0.3141)
```

```
>>> WMAP9.Ode0, newcosmo.Ode0  # Indirectly changed since this is
flat
(0.7134130719051658, 0.6858130719051657)
```

Finding the Redshift at a Given Value of a Cosmological Quantity

If you know a cosmological quantity and you want to know the redshift which it corresponds to, you can use `z_at_value`.

Example

To find the redshift using `z_at_value`:

```
>>> import astropy.units as u
>>> from astropy.cosmology import Planck13, z_at_value
>>> z_at_value(Planck13.age, 2 * u.Gyr)
3.1981226843560968
```

For some quantities, there can be more than one redshift that satisfies a value. In this case you can use the `zmin` and `zmax` keywords to restrict the search range. See the `z_at_value` docstring for more detailed usage examples.

Built-in Cosmologies

A number of preloaded cosmologies are available from analyses using the WMAP and Planck satellite data. For example:

```
>>> from astropy.cosmology import Planck13  # Planck 2013
>>> Planck13.lookback_time(2)  # lookback time in Gyr at z=2
<Quantity 10.51184138 Gyr>
```

A full list of the predefined cosmologies is given by `cosmology.parameters.available` and summarized below:

Name	Source	H0	Om	Flat
WMAP5	Komatsu et al. 2009	70.2	0.277	Yes
WMAP7	Komatsu et al. 2011	70.4	0.272	Yes
WMAP9	Hinshaw et al. 2013	69.3	0.287	Yes
Planck13	Planck Collab 2013, Paper XVI	67.8	0.307	Yes
Planck15	Planck Collab 2015, Paper XIII	67.7	0.307	Yes
Planck18	Planck Collab 2018, Paper VI	67.7	0.310	Yes

Note

Unlike the Planck 2015 paper, the Planck 2018 paper includes massive neutrinos in `Om0` but the Planck18 object includes them in `m_nu` instead for consistency. Hence, the `Om0` value in Planck18 differs slightly from the Planck 2018 paper but represents the same cosmological model.

Currently, all are instances of **`FlatLambdaCDM`**. More details about exactly where each set of parameters comes from are available in the docstring for each object:

```
>>> from astropy.cosmology import WMAP7
>>> print(WMAP7.__doc__)
WMAP7 instance of FlatLambdaCDM cosmology
(from Komatsu et al. 2011, ApJS, 192, 18, doi: 10.1088/0067-0049
/192/2/18.
Table 1 (WMAP + BAO + H0 ML).)
```

Specifying a Dark Energy Model

Along with the standard **`FlatLambdaCDM`** model described above, a number of additional dark energy models are provided. **`FlatLambdaCDM`** and **`LambdaCDM`** assume that dark energy is a cosmological constant, and should be the most commonly used cases; the former assumes a flat universe, the latter allows for spatial curvature. **`FlatwCDM`** and **`wCDM`** assume a constant dark energy equation of state parameterized by \(w_{0}\). Two forms of a variable dark energy equation of state are provided: the simple first order linear expansion \(w(z) = w_{0} + w_{z} z\) by **`w0wzCDM`**, as well as the common CPL form by **`w0waCDM`**: \(w(z) = w_{0} + w_{a} (1 - a) = w_{0} + w_{a} z / (1 + z)\) and its generalization to include a pivot redshift by **`wpwaCDM`**: \(w(z) = w_{p} + w_{a} (a_{p} - a)\).

Users can specify their own equation of state by subclassing **`FLRW`**. See the provided subclasses for examples. It is recommended, but not required, that all arguments to the constructor of a new subclass be available as properties, since the `clone` method assumes this is the case. It is also advisable to stick to subclassing **`FLRW`** rather than one of its subclasses, since some of them use internal optimizations that also need to be propagated to any subclasses. Users wishing to use similar tricks (which can make distance calculations much faster) should consult the cosmology module source code for details.

Photons and Neutrinos

The cosmology classes (can) include the contribution to the energy density from both photons and neutrinos. By default, the latter are assumed massless. The three parameters controlling the properties of these species, which are arguments to the initializers of all of the cosmological classes, are `Tcmb0` (the temperature of the cosmic microwave background at z=0), `Neff` (the effective number of neutrino species), and `m_nu` (the rest mass of the neutrino species). `Tcmb0` and `m_nu` should be expressed as unit Quantities. All three have standard default values — 0 K, 3.04, and 0 eV, respectively. (The reason that `Neff` is not 3 has to do primarily with a small bump in the neutrino energy spectrum due to electron- positron annihilation, but is also affected by weak interaction physics.) Setting the CMB temperature to 0 removes the contribution of both neutrinos and photons. This is the default to ensure these components are excluded unless the user explicitly requests them.

Massive neutrinos are treated using the approach described in the WMAP seven-year cosmology paper (Komatsu et al. 2011, ApJS, 192, 18, section 3.3). This is not the simple \(\Omega_{\nu 0} h^2 = \sum_i m_{\nu\, i} / 93.04\, \mathrm{eV}\) approximation. Also note that the values of \(\Omega_{\nu}(z)\) include both the kinetic energy and the rest mass energy components, and that the Planck13 and Planck15 cosmologies include a single species of neutrinos with non-zero mass (which is not included in \(\Omega_{m0}\)).

Adding massive neutrinos can have significant performance implications. In particular, the computation of distance measures and lookback times are factors of three to four times slower than in the massless neutrino case. Therefore, if you need to compute many distances in such a cosmology and performance is critical, it is particularly useful to calculate them on a grid and use interpolation.

Examples

The contribution of photons and neutrinos to the total mass-energy density can be found as a function of redshift:

```
>>> from astropy.cosmology import WMAP7   # WMAP 7-year cosmology
>>> WMAP7.Ogamma0, WMAP7.Onu0  # Current epoch values
(4.985694972799396e-05, 3.442154948307989e-05)
>>> z = [0, 1.0, 2.0]
>>> WMAP7.Ogamma(z), WMAP7.Onu(z)
(array([4.98603986e-05, 2.74593395e-04, 4.99915942e-04]),
 array([3.44239306e-05, 1.89580995e-04, 3.45145089e-04]))
```

If you want to exclude photons and neutrinos from your calculations, you can set `Tcmb0` to 0 (which is also the default):

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> import astropy.units as u
>>> cos = FlatLambdaCDM(70.4 * u.km / u.s / u.Mpc, 0.272, Tcmb0 = 0.0
* u.K)
>>> cos.Ogamma0, cos.Onu0
(0.0, 0.0)
```

You can include photons but exclude any contributions from neutrinos by setting `Tcmb0` to be non-zero (2.725 K is the standard value for our Universe) but setting `Neff` to 0:

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> cos = FlatLambdaCDM(70.4, 0.272, Tcmb0=2.725, Neff=0)
>>> cos.Ogamma([0, 1, 2])  # Photons are still present
array([4.98603986e-05, 2.74642208e-04, 5.00086413e-04])
>>> cos.Onu([0, 1, 2])  # But not neutrinos
array([0., 0., 0.])
```

The number of neutrino species is assumed to be the floor of `Neff`, which in the default case is `Neff=3`. Therefore, if non-zero neutrino masses are desired, then three masses should be provided. However, if only one value is provided, all of the species are assumed to have the same mass. `Neff` is assumed to be shared equally between each species.

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> import astropy.units as u
>>> H0 = 70.4 * u.km / u.s / u.Mpc
>>> m_nu = 0 * u.eV
>>> cosmo = FlatLambdaCDM(H0, 0.272, Tcmb0=2.725, m_nu=m_nu)
>>> cosmo.has_massive_nu
False
>>> cosmo.m_nu
<Quantity [0., 0., 0.] eV>
>>> m_nu = [0.0, 0.05, 0.10] * u.eV
>>> cosmo = FlatLambdaCDM(H0, 0.272, Tcmb0=2.725, m_nu=m_nu)
>>> cosmo.has_massive_nu
True
>>> cosmo.m_nu
<Quantity [0.  , 0.05, 0.1 ] eV>
>>> cosmo.Onu([0, 1.0, 15.0])
array([0.00327011, 0.00896845, 0.01257946])
>>> cosmo.Onu(1) * cosmo.critical_density(1)
<Quantity 2.444380380370406e-31 g / cm3>
```

While these examples used **`FlatLambdaCDM`**, the above examples also apply for all of the other cosmology classes.

For Developers: Using `astropy.cosmology` Inside `astropy`

If you are writing code for the `astropy` core or an affiliated package, it is often useful to assume a default cosmology so that the exact cosmology does not have to be specified every time a function or method is called. In this case, it is possible to specify a "default" cosmology.

You can set the default cosmology to a predefined value by using the "default_cosmology" option in the `[cosmology.core]` section of the configuration file (see Configuration System (astropy.config)). Alternatively, you can use the `set` function of **`default_cosmology`** to set a cosmology for the current Python session. If you have not set a default cosmology using one of the methods described above, then the cosmology module will default to using the nine-year WMAP parameters.

It is strongly recommended that you use the default cosmology through the **`default_cosmology`** science state object. An override option can then be provided using something like the following:

```
def myfunc(..., cosmo=None):
    from astropy.cosmology import default_cosmology

    if cosmo is None:
        cosmo = default_cosmology.get()

    ... your code here ...
```

This ensures that all code consistently uses the default cosmology unless explicitly overridden.

Note

In general it is better to use an explicit cosmology (for example `WMAP9.H(0)` instead of `cosmology.default_cosmology.get().H(0)`). Use of the default cosmology should generally be reserved for code that will be included in the `astropy` core or an affiliated package.

See Also

- Hogg, "Distance measures in cosmology", https://arxiv.org/abs/astro-ph/9905116
- Linder, "Exploring the Expansion History of the Universe", https://arxiv.org/abs/astro-ph/0208512
- NASA's Legacy Archive for Microwave Background Data Analysis,

https://lambda.gsfc.nasa.gov/

Range of Validity and Reliability

The code in this sub-package is tested against several widely used online cosmology calculators and has been used to perform many calculations in refereed papers. You can check the range of redshifts over which the code is regularly tested in the module `astropy.cosmology.tests.test_cosmology`. If you find any bugs, please let us know by opening an issue at the GitHub repository!

A more difficult question is the range of redshifts over which the code is expected to return valid results. This is necessarily model-dependent, but in general you should not expect the numeric results to be well behaved for redshifts more than a few times larger than the epoch of matter-radiation equality (so, for typical models, not above z = 5-6,000, but for some models much lower redshifts may be ill-behaved). In particular, you should pay attention to warnings from the `scipy` integration package about integrals failing to converge (which may only be issued once per session).

The built-in cosmologies use the parameters as listed in the respective papers. These provide only a limited range of precision, and so you should not expect derived quantities to match beyond that precision. For example, the Planck 2013 and 2015 results only provide the Hubble constant to four digits. Therefore, they should not be expected to match the age quoted by the Planck team to better than that, despite the fact that five digits are quoted in the papers.

Reference/API

astropy.cosmology Package

astropy.cosmology contains classes and functions for cosmological distance measures and other cosmology-related calculations.

See the Astropy documentation for more detailed usage examples and references.

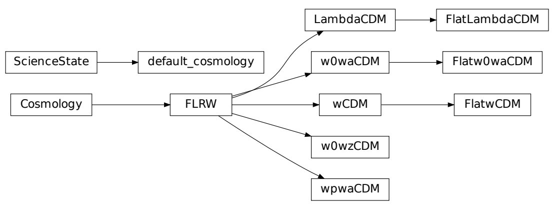
Functions

`z_at_value`(func, fval[, zmin, zmax, ztol, …])	Find the redshift `z` at which `func(z) = fval`.

Classes

FLRW(H0, Om0, Ode0[, Tcmb0, Neff, m_nu, …])	A class describing an isotropic and homogeneous (Friedmann-Lemaitre-Robertson-Walker) cosmology.
FlatLambdaCDM(H0, Om0[, Tcmb0, Neff, m_nu, …])	FLRW cosmology with a cosmological constant and no curvature.
Flatw0waCDM(H0, Om0[, w0, wa, Tcmb0, Neff, …])	FLRW cosmology with a CPL dark energy equation of state and no curvature.
FlatwCDM(H0, Om0[, w0, Tcmb0, Neff, m_nu, …])	FLRW cosmology with a constant dark energy equation of state and no spatial curvature.
LambdaCDM(H0, Om0, Ode0[, Tcmb0, Neff, …])	FLRW cosmology with a cosmological constant and curvature.
default_cosmology()	The default cosmology to use. To change it::.
w0waCDM(H0, Om0, Ode0[, w0, wa, Tcmb0, …])	FLRW cosmology with a CPL dark energy equation of state and curvature.
w0wzCDM(H0, Om0, Ode0[, w0, wz, Tcmb0, …])	FLRW cosmology with a variable dark energy equation of state and curvature.
wCDM(H0, Om0, Ode0[, w0, Tcmb0, Neff, m_nu, …])	FLRW cosmology with a constant dark energy equation of state and curvature.
wpwaCDM(H0, Om0, Ode0[, wp, wa, zp, Tcmb0, …])	FLRW cosmology with a CPL dark energy equation of state, a pivot redshift, and curvature.

Class Inheritance Diagram



Convolution and Filtering (`astropy.convolution`)

Introduction

`astropy.convolution` provides convolution functions and kernels that offer improvements compared to the SciPy `scipy.ndimage` convolution routines, including:

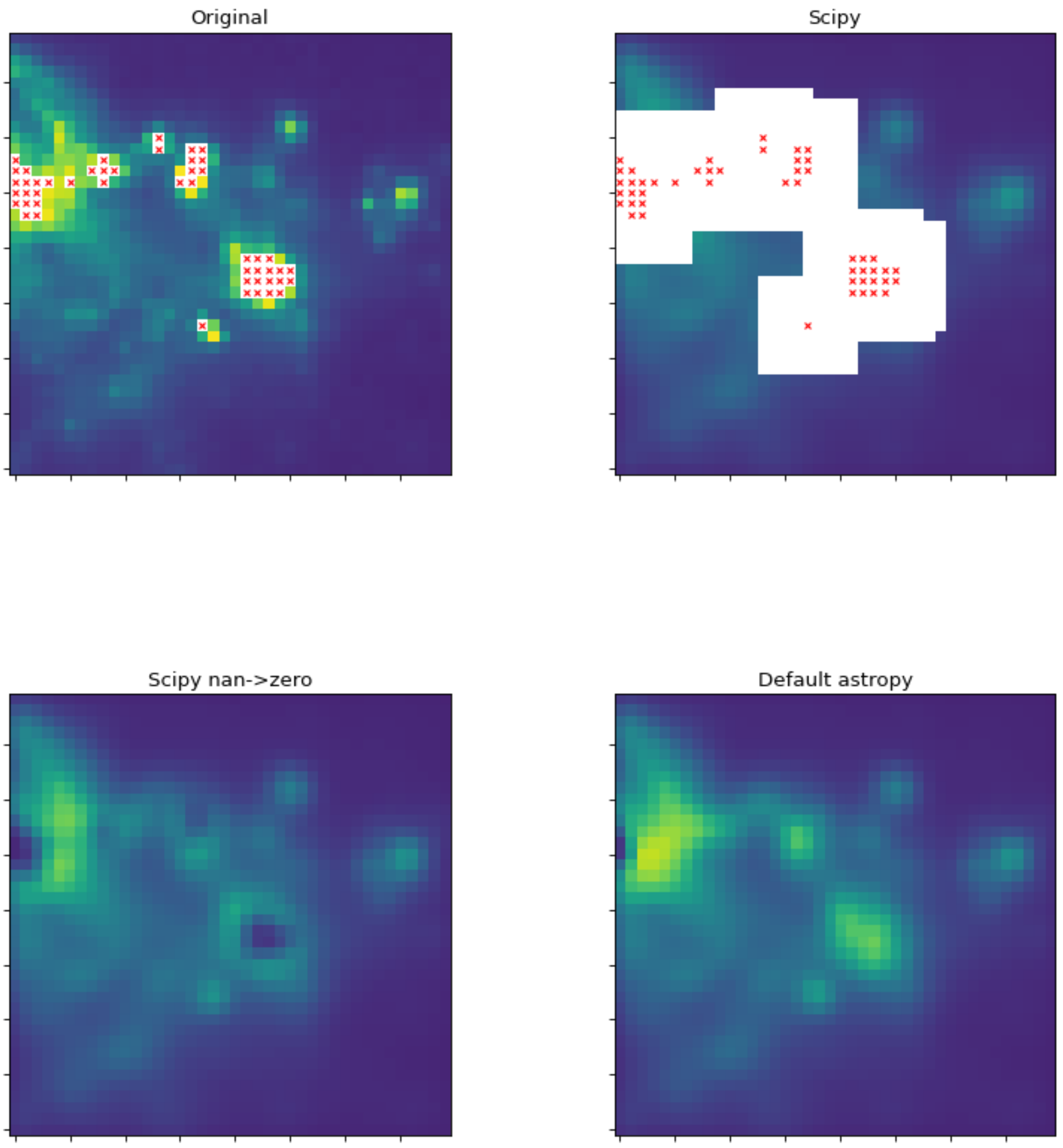
- Proper treatment of NaN values (ignoring them during convolution and

replacing NaN pixels with interpolated values)
- A single function for 1D, 2D, and 3D convolution
- Improved options for the treatment of edges
- Both direct and Fast Fourier Transform (FFT) versions
- Built-in kernels that are commonly used in Astronomy

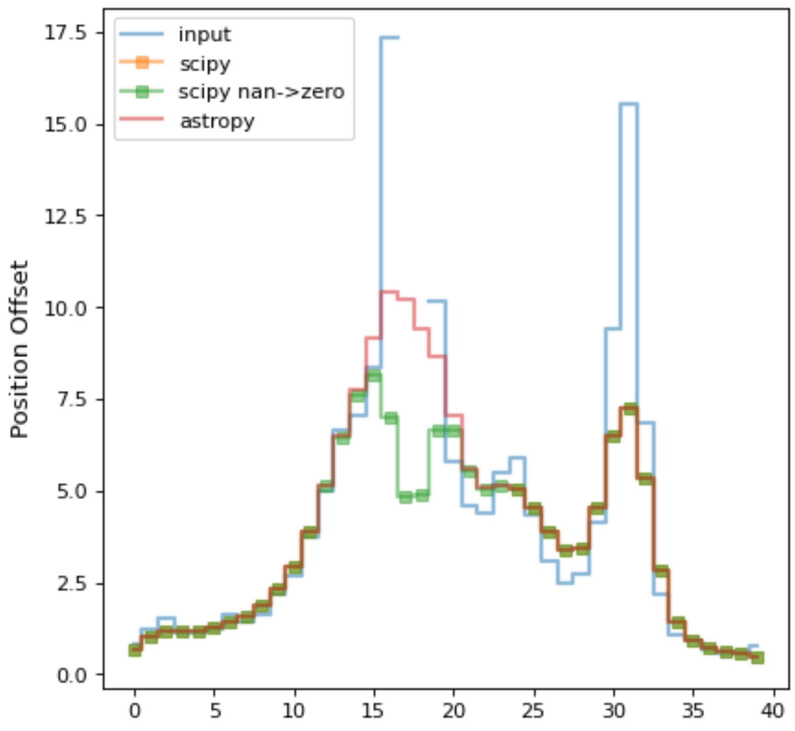
The following thumbnails show the difference between `scipy` and `astropy` convolve functions on an astronomical image that contains NaN values. `scipy` 's function essentially returns NaN for all pixels that are within a kernel of any NaN value, which is often not the desired result.

```
import numpy as np
import matplotlib.pyplot as plt

from astropy.io import fits
from astropy.utils.data import get_pkg_data_filename
from astropy.convolution import Gaussian2DKernel
from scipy.signal import convolve as scipy_convolve
from astropy.convolution import convolve


# Load the data from data.astropy.org
filename = get_pkg_data_filename('galactic_center/gc_msx_e.fits')
hdu = fits.open(filename)[0]

# Scale the file to have reasonable numbers
# (this is mostly so that colorbars do not have too many digits)
# Also, we crop it so you can see individual pixels
img = hdu.data[50:90, 60:100] * 1e5

# This example is intended to demonstrate how astropy.convolve and
# scipy.convolve handle missing data, so we start by setting the
# brightest pixels to NaN to simulate a "saturated" data set
img[img > 2e1] = np.nan

# We also create a copy of the data and set those NaNs to zero.  We
will
# use this for the scipy convolution
img_zerod = img.copy()
img_zerod[np.isnan(img)] = 0

# We smooth with a Gaussian kernel with x_stddev=1 (and y_stddev=1)
# It is a 9x9 array
kernel = Gaussian2DKernel(x_stddev=1)

# Convolution: scipy's direct convolution mode spreads out NaNs (see
# panel 2 below)
```

```
scipy_conv = scipy_convolve(img, kernel, mode='same',
method='direct')

# scipy's direct convolution mode run on the 'zero'd' image will not
# have NaNs, but will have some very low value zones where the NaNs
were
# (see panel 3 below)
scipy_conv_zerod = scipy_convolve(img_zerod, kernel, mode='same',
                                  method='direct')

# astropy's convolution replaces the NaN pixels with a kernel-
weighted
# interpolation from their neighbors
astropy_conv = convolve(img, kernel)


# Now we do a bunch of plots.  In the first two plots, the originally
masked
# values are marked with red X's
plt.figure(1, figsize=(12, 12)).clf()
ax1 = plt.subplot(2, 2, 1)
im = ax1.imshow(img, vmin=-2., vmax=2.e1, origin='lower',
                interpolation='nearest', cmap='viridis')
y, x = np.where(np.isnan(img))
ax1.set_autoscale_on(False)
ax1.plot(x, y, 'rx', markersize=4)
ax1.set_title("Original")
ax1.set_xticklabels([])
ax1.set_yticklabels([])

ax2 = plt.subplot(2, 2, 2)
im = ax2.imshow(scipy_conv, vmin=-2., vmax=2.e1, origin='lower',
                interpolation='nearest', cmap='viridis')
ax2.set_autoscale_on(False)
ax2.plot(x, y, 'rx', markersize=4)
ax2.set_title("Scipy")
ax2.set_xticklabels([])
ax2.set_yticklabels([])

ax3 = plt.subplot(2, 2, 3)
im = ax3.imshow(scipy_conv_zerod, vmin=-2., vmax=2.e1,
origin='lower',
                interpolation='nearest', cmap='viridis')
ax3.set_title("Scipy nan->zero")
ax3.set_xticklabels([])
ax3.set_yticklabels([])

ax4 = plt.subplot(2, 2, 4)
im = ax4.imshow(astropy_conv, vmin=-2., vmax=2.e1, origin='lower',
                interpolation='nearest', cmap='viridis')
ax4.set_title("Default astropy")
ax4.set_xticklabels([])
ax4.set_yticklabels([])

# we make a second plot of the amplitudes vs offset position to more
# clearly illustrate the value differences
plt.figure(2).clf()
plt.plot(img[:, 25], label='input', drawstyle='steps-mid',
linewidth=2,
         alpha=0.5)
plt.plot(scipy_conv[:, 25], label='scipy', drawstyle='steps-mid',
         linewidth=2, alpha=0.5, marker='s')
plt.plot(scipy_conv_zerod[:, 25], label='scipy nan->zero',
         drawstyle='steps-mid', linewidth=2, alpha=0.5, marker='s')
plt.plot(astropy_conv[:, 25], label='astropy', drawstyle='steps-mid',
         linewidth=2, alpha=0.5)
plt.ylabel("Amplitude")
plt.ylabel("Position Offset")
plt.legend(loc='best')
plt.show()
```



(png, svg, pdf)



(png, svg, pdf)

The following sections describe how to make use of the convolution functions, and how to use built-in convolution kernels:

Getting Started

Two convolution functions are provided. They are imported as:

```
from astropy.convolution import convolve, convolve_fft
```

and are both used as:

```
result = convolve(image, kernel)
result = convolve_fft(image, kernel)
```

`convolve()` is implemented as a direct convolution algorithm, while `convolve_fft()` uses a Fast Fourier Transform (FFT). Thus, the former is better for small kernels, while the latter is much more efficient for larger kernels.

Example

To convolve a 1D dataset with a user-specified kernel, you can do:

```
>>> from astropy.convolution import convolve
>>> convolve([1, 4, 5, 6, 5, 7, 8], [0.2, 0.6, 0.2])
array([1.4, 3.6, 5. , 5.6, 5.6, 6.8, 6.2])
```

Notice that the end points are set to zero — by default, points that are too close to the boundary to have a convolved value calculated are set to zero. However, the `convolve()` function allows for a `boundary` argument that can be used to specify alternate behaviors. For example, setting `boundary='extend'` causes values near the edges to be computed, assuming the original data is simply extended using a constant extrapolation beyond the boundary:

```
>>> from astropy.convolution import convolve
>>> convolve([1, 4, 5, 6, 5, 7, 8], [0.2, 0.6, 0.2],
boundary='extend')
array([1.6, 3.6, 5. , 5.6, 5.6, 6.8, 7.8])
```

The values at the end are computed assuming that any value below the first point is `1`, and any value above the last point is `8`. For a more detailed discussion of boundary treatment, see Using the Convolution Functions.

Example

The convolution module also includes built-in kernels that can be imported as, for example:

```
>>> from astropy.convolution import Gaussian1DKernel
```

To use a kernel, first create a specific instance of the kernel:

```
>>> gauss = Gaussian1DKernel(stddev=2)
```

`gauss` is not an array, but a kernel object. The underlying array can be retrieved with:

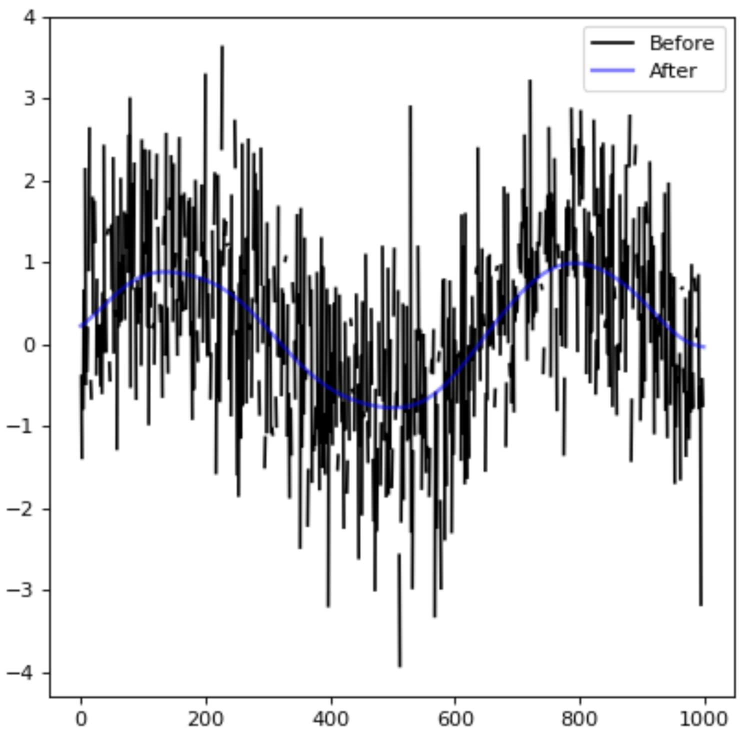
```
>>> gauss.array
array([6.69151129e-05, 4.36341348e-04, 2.21592421e-03,
       8.76415025e-03, 2.69954833e-02, 6.47587978e-02,
       1.20985362e-01, 1.76032663e-01, 1.99471140e-01,
       1.76032663e-01, 1.20985362e-01, 6.47587978e-02,
       2.69954833e-02, 8.76415025e-03, 2.21592421e-03,
       4.36341348e-04, 6.69151129e-05])
```

The kernel can then be used directly when calling `convolve()`:

```
import numpy as np
import matplotlib.pyplot as plt

from astropy.convolution import Gaussian1DKernel, convolve
```

```
plt.figure(3).clf()

# Generate fake data
x = np.arange(1000).astype(float)
y = np.sin(x / 100.) + np.random.normal(0., 1., x.shape)
y[::3] = np.nan

# Create kernel
g = Gaussian1DKernel(stddev=50)

# Convolve data
z = convolve(y, g)

# Plot data before and after convolution
plt.plot(x, y, 'k-', label='Before')
plt.plot(x, z, 'b-', label='After', alpha=0.5, linewidth=2)
plt.legend(loc='best')
plt.show()
```

(png, svg, pdf)



Using `astropy`'s Convolution to Replace Bad Data

`astropy`'s convolution methods can be used to replace bad data with values interpolated from their neighbors. Kernel-based interpolation is useful for handling images with a few bad pixels or for interpolating sparsely sampled images.

The interpolation tool is implemented and used as:

```
from astropy.convolution import interpolate_replace_nans
result = interpolate_replace_nans(image, kernel)
```

Some contexts in which you might want to use kernel-based interpolation include:

- Images with saturated pixels. Generally, these are the highest-intensity regions in the imaged area, and the interpolated values are not reliable, but this can be useful for display purposes.
- Images with flagged pixels (e.g., a few small regions affected by cosmic rays or other spurious signals that require those pixels to be flagged out). If the affected region is small enough, the resulting interpolation will have a small effect on source statistics and may allow for robust source-finding algorithms to be run on the resulting data.
- Sparsely sampled images such as those constructed with single-pixel detectors. Such images will only have a few discrete points sampled across the imaged area, but an approximation of the extended sky emission can still be constructed.

Note

Care must be taken to ensure that the kernel is large enough to completely cover potential contiguous regions of NaN values. An `AstropyUserWarning` is raised if NaN values are detected post-convolution, in which case the kernel size should be increased.

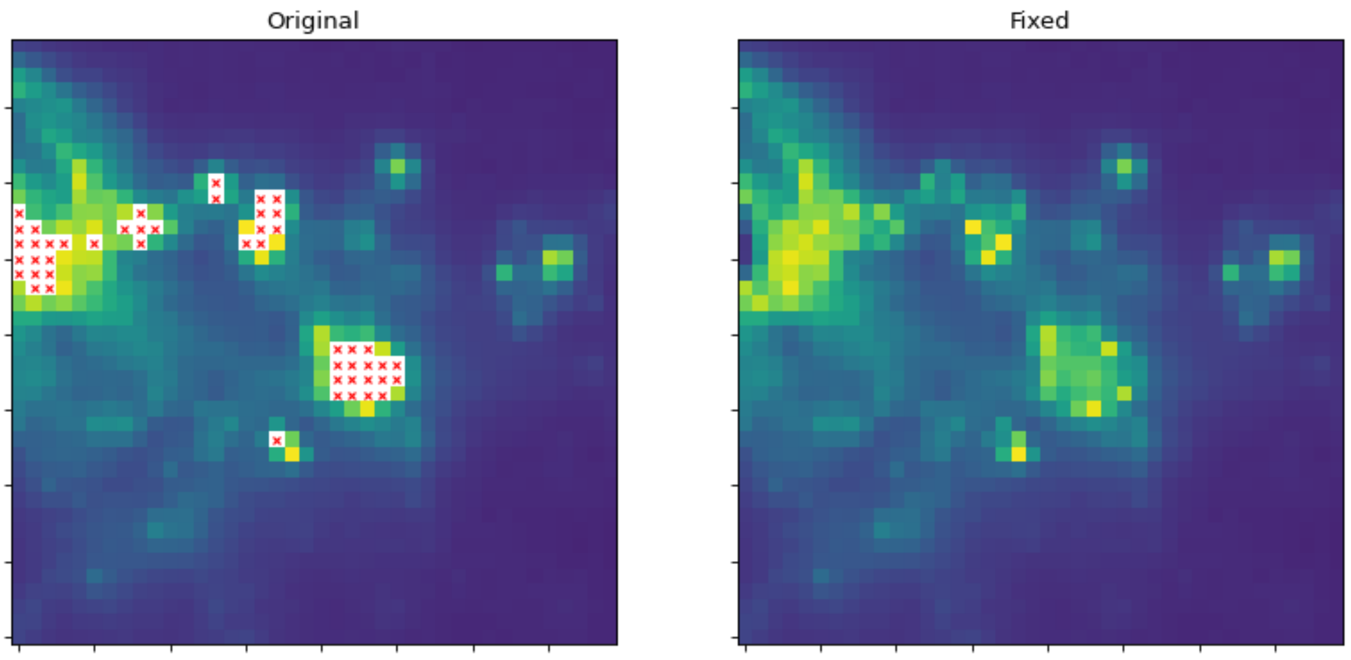
Example

The script below shows an example of kernel interpolation to fill in flagged-out pixels:

```
import numpy as np
import matplotlib.pyplot as plt

from astropy.io import fits
from astropy.utils.data import get_pkg_data_filename
from astropy.convolution import Gaussian2DKernel,
interpolate_replace_nans

# Load the data from data.astropy.org
filename = get_pkg_data_filename('galactic_center/gc_msx_e.fits')
```

```
hdu = fits.open(filename)[0]
img = hdu.data[50:90, 60:100] * 1e5

# This example is intended to demonstrate how astropy.convolve and
# scipy.convolve handle missing data, so we start by setting the
brightest
# pixels to NaN to simulate a "saturated" data set
img[img > 2e1] = np.nan

# We smooth with a Gaussian kernel with x_stddev=1 (and y_stddev=1)
# It is a 9x9 array
kernel = Gaussian2DKernel(x_stddev=1)

# create a "fixed" image with NaNs replaced by interpolated values
fixed_image = interpolate_replace_nans(img, kernel)

# Now we do a bunch of plots.  In the first two plots, the originally
masked
# values are marked with red X's
plt.figure(1, figsize=(12, 6)).clf()
plt.close(2) # close the second plot from above

ax1 = plt.subplot(1, 2, 1)
im = ax1.imshow(img, vmin=-2., vmax=2.e1, origin='lower',
                interpolation='nearest', cmap='viridis')
y, x = np.where(np.isnan(img))
ax1.set_autoscale_on(False)
ax1.plot(x, y, 'rx', markersize=4)
ax1.set_title("Original")
ax1.set_xticklabels([])
ax1.set_yticklabels([])

ax2 = plt.subplot(1, 2, 2)
im = ax2.imshow(fixed_image, vmin=-2., vmax=2.e1, origin='lower',
                interpolation='nearest', cmap='viridis')
ax2.set_title("Fixed")
ax2.set_xticklabels([])
ax2.set_yticklabels([])
```

(png, svg, pdf)



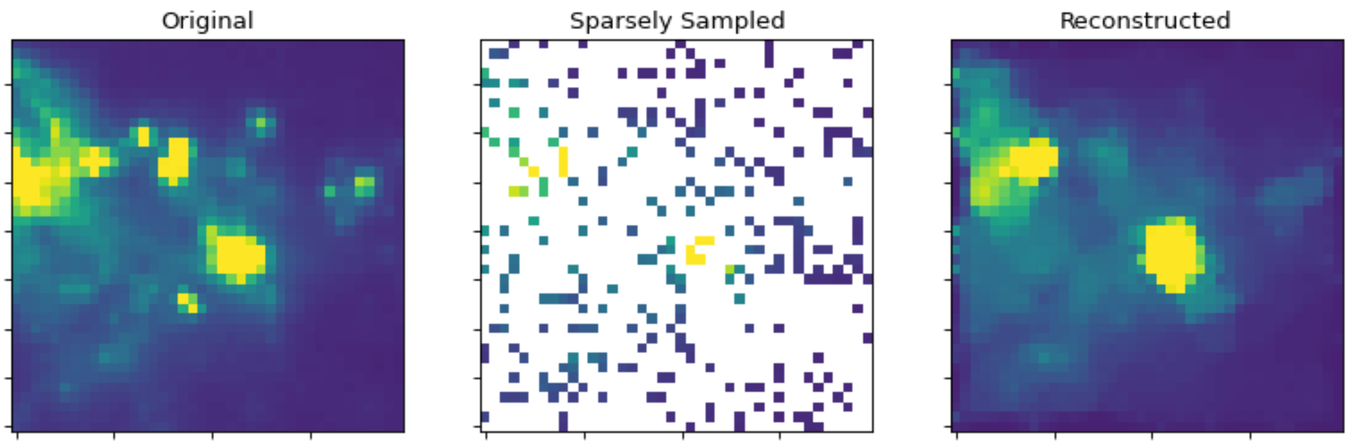
Example

This script shows the power of this technique for reconstructing images from sparse sampling. Note that the image is not perfect: the pointlike sources are sometimes missed, but the extended structure is very well recovered by eye.

```
import numpy as np
import matplotlib.pyplot as plt

from astropy.io import fits
from astropy.utils.data import get_pkg_data_filename
from astropy.convolution import Gaussian2DKernel,
interpolate_replace_nans

# Load the data from data.astropy.org
filename = get_pkg_data_filename('galactic_center/gc_msx_e.fits')

hdu = fits.open(filename)[0]
img = hdu.data[50:90, 60:100] * 1e5

indices = np.random.randint(low=0, high=img.size, size=300)

sampled_data = img.flat[indices]

# Build a new, sparsely sampled version of the original image
new_img = np.tile(np.nan, img.shape)
```

```
new_img.flat[indices] = sampled_data

# We smooth with a Gaussian kernel with x_stddev=1 (and y_stddev=1)
# It is a 9x9 array
kernel = Gaussian2DKernel(x_stddev=1)

# create a "reconstructed" image with NaNs replaced by interpolated
values
reconstructed_image = interpolate_replace_nans(new_img, kernel)

# Now we do a bunch of plots.  In the first two plots, the originally
masked
# values are marked with red X's
plt.figure(1, figsize=(12, 6)).clf()
ax1 = plt.subplot(1, 3, 1)
im = ax1.imshow(img, vmin=-2., vmax=2.e1, origin='lower',
                interpolation='nearest', cmap='viridis')
y, x = np.where(np.isnan(img))
ax1.set_autoscale_on(False)
ax1.set_title("Original")
ax1.set_xticklabels([])
ax1.set_yticklabels([])

ax2 = plt.subplot(1, 3, 2)
im = ax2.imshow(new_img, vmin=-2., vmax=2.e1, origin='lower',
                interpolation='nearest', cmap='viridis')
ax2.set_title("Sparsely Sampled")
ax2.set_xticklabels([])
ax2.set_yticklabels([])

ax2 = plt.subplot(1, 3, 3)
im = ax2.imshow(reconstructed_image, vmin=-2., vmax=2.e1,
origin='lower',
                interpolation='nearest', cmap='viridis')
ax2.set_title("Reconstructed")
ax2.set_xticklabels([])
ax2.set_yticklabels([])
```

(png, svg, pdf)



A Note on Backward Compatibility (pre v2.0)

The behavior of `astropy`'s direct convolution (`convolve()`) changed in version 2.0. Generally, the old version is undesirable. However, to recover the behavior of the old (`astropy` version <2.0) direct convolution function, you can interpolate and then convolve, for example:

```
from astropy.convolution import interpolate_replace_nans, convolve
interped_result = interpolate_replace_nans(image, kernel)
result = convolve(interped_image, kernel)
```

Note that the default behavior of both `convolve` and `convolve_fft` is to perform *normalized convolution* and interpolate NaNs during that process. The example given in this note, and what was previously done only in direct convolution in old versions of `astropy` now does a two-step process: first, it replaces the NaNs with their interpolated values while leaving all non-NaN values unchanged, then it convolves the resulting image with the specified kernel.

Using `astropy.convolution`

Using the Convolution Functions

Overview

Two convolution functions are provided. They are imported as:

```
>>> from astropy.convolution import convolve, convolve_fft
```

and are both used as:

```
>>> result = convolve(image, kernel)
>>> result = convolve_fft(image, kernel)
```

`convolve()` is implemented as a direct convolution algorithm, while `convolve_fft()` uses a Fast Fourier Transform (FFT). Thus, the former is better for small kernels, while the latter is much more efficient for larger kernels.

The input images and kernels should be lists or `numpy` arrays with either 1, 2, or 3 dimensions (and the number of dimensions should be the same for the image and kernel). The result is a `numpy` array with the same dimensions as the input image. The convolution is always done as floating point.

The `convolve()` function takes an optional `boundary=` argument describing how to perform the convolution at the edge of the array. The values for `boundary` can be:

- `None` : set the result values to zero where the kernel extends beyond the edge of the array (default).
- `'fill'` : set values outside the array boundary to a constant. If this option is specified, the constant should be specified using the `fill_value=` argument, which defaults to zero.
- `'wrap'` : assume that the boundaries are periodic.
- `'extend'` : set values outside the array to the nearest array value.

By default, the kernel is not normalized. To normalize it prior to convolution, use:

```
>>> result = convolve(image, kernel, normalize_kernel=True)
```

Examples

Smooth a 1D array with a custom kernel and no boundary treatment:

```
>>> import numpy as np
>>> convolve([1, 4, 5, 6, 5, 7, 8], [0.2, 0.6, 0.2])
array([1.4, 3.6, 5. , 5.6, 5.6, 6.8, 6.2])
```

As above, but using the ‘extend’ algorithm for boundaries:

```
>>> convolve([1, 4, 5, 6, 5, 7, 8], [0.2, 0.6, 0.2],
boundary='extend')
array([1.6, 3.6, 5. , 5.6, 5.6, 6.8, 7.8])
```

If a NaN value is present in the original array, it will be interpolated using the kernel:

```
>>> import numpy as np
>>> convolve([1, 4, 5, 6, np.nan, 7, 8], [0.2, 0.6, 0.2],
boundary='extend')
array([1.6 , 3.6 , 5.  , 5.75, 6.5 , 7.25, 7.8 ])
```

Kernels and arrays can be specified either as lists or as `numpy` arrays. The following examples show how to construct a 1D array as a list:

```
>>> kernel = [0, 1, 0]
>>> result = convolve(spectrum, kernel)
```

A 2D array as a list:

```
>>> kernel = [[0, 1, 0],
...           [1, 2, 1],
...           [0, 1, 0]]
>>> result = convolve(image, kernel)
```

And a 3D array as a list:

```
>>> kernel = [[[0, 0, 0], [0, 2, 0], [0, 0, 0]],
...           [[0, 1, 0], [2, 3, 2], [0, 1, 0]],
...           [[0, 0, 0], [0, 2, 0], [0, 0, 0]]]
>>> result = convolve(cube, kernel)
```

Kernels

The above examples use custom kernels, but **`astropy.convolution`** also includes a number of built-in kernels, which are described in Convolution Kernels.

Convolution Kernels

Introduction and Concept

The convolution module provides several built-in kernels to cover the most common applications in astronomy. It is also possible to define custom kernels from arrays or combine existing kernels to match specific applications.

Every filter kernel is characterized by its response function. For time series we speak of an “impulse response function” or for images we call it “point spread function.” This response function is given for every kernel by a **`FittableModel`**, which is evaluated on a grid with **`discretize_model()`** to obtain a kernel array, which can be used for discrete convolution with the binned data.

Examples

1D Kernels

One application of filtering is to smooth noisy data. In this case we consider a noisy Lorentz curve:

```
>>> import numpy as np
>>> from astropy.modeling.models import Lorentz1D
>>> from astropy.convolution import convolve, Gaussian1DKernel,
Box1DKernel
>>> lorentz = Lorentz1D(1, 0, 1)
>>> x = np.linspace(-5, 5, 100)
>>> data_1D = lorentz(x) + 0.1 * (np.random.rand(100) - 0.5)
```

Smoothing the noisy data with a **`Gaussian1DKernel`** with a standard deviation of 2 pixels:

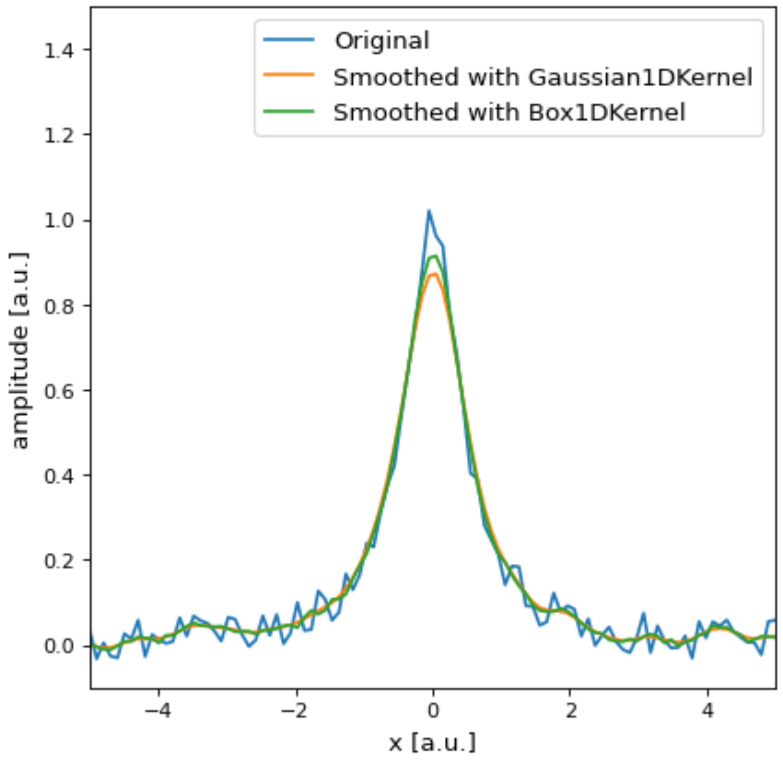
```
>>> gauss_kernel = Gaussian1DKernel(2)
>>> smoothed_data_gauss = convolve(data_1D, gauss_kernel)
```

Smoothing the same data with a **`Box1DKernel`** of width 5 pixels:

```
>>> box_kernel = Box1DKernel(5)
>>> smoothed_data_box = convolve(data_1D, box_kernel)
```

The following plot illustrates the results:

(png, svg, pdf)



Beside the `astropy` convolution functions **convolve** and **convolve_fft**, it is also possible to use the kernels with `numpy` or `scipy` convolution by passing the `array` attribute. This will be faster in most cases than the `astropy` convolution, but will not work properly if NaN values are present in the data.

```
>>> smoothed = np.convolve(data_1D, box_kernel.array)
```

2D Kernels

As all 2D kernels are symmetric, it is sufficient to specify the width in one direction. Therefore the use of 2D kernels is basically the same as for 1D kernels. Here we consider a small Gaussian-shaped source of amplitude 1 in the middle of the image and add 10% noise:

```
>>> import numpy as np
>>> from astropy.convolution import convolve, Gaussian2DKernel,
Tophat2DKernel
>>> from astropy.modeling.models import Gaussian2D
>>> gauss = Gaussian2D(1, 0, 0, 3, 3)
>>> # Fake image data including noise
>>> x = np.arange(-100, 101)
>>> y = np.arange(-100, 101)
>>> x, y = np.meshgrid(x, y)
>>> data_2D = gauss(x, y) + 0.1 * (np.random.rand(201, 201) - 0.5)
```

Smoothing the noisy data with a **`Gaussian2DKernel`** with a standard deviation of 2 pixels:

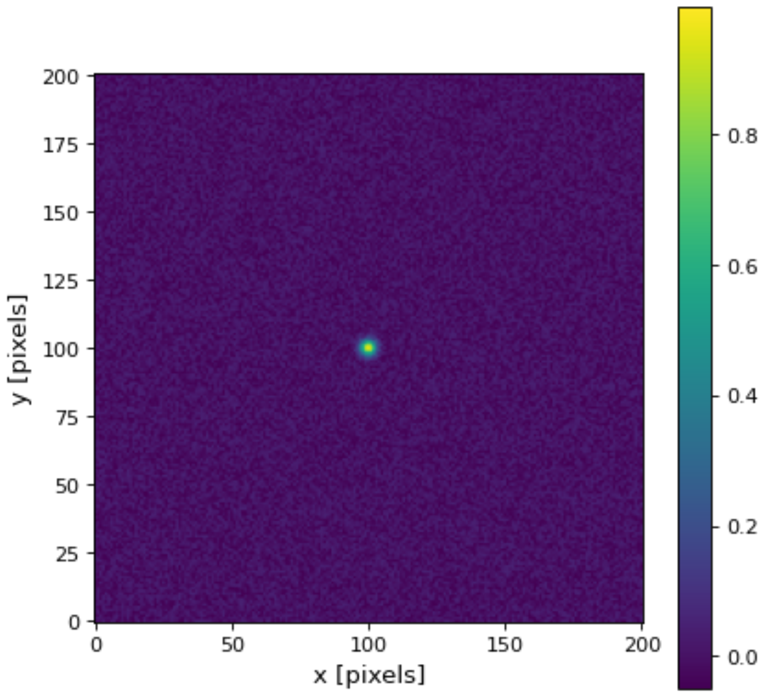
```
>>> gauss_kernel = Gaussian2DKernel(2)
>>> smoothed_data_gauss = convolve(data_2D, gauss_kernel)
```

Smoothing the noisy data with a **`Tophat2DKernel`** of width 5 pixels:

```
>>> tophat_kernel = Tophat2DKernel(5)
>>> smoothed_data_tophat = convolve(data_2D, tophat_kernel)
```

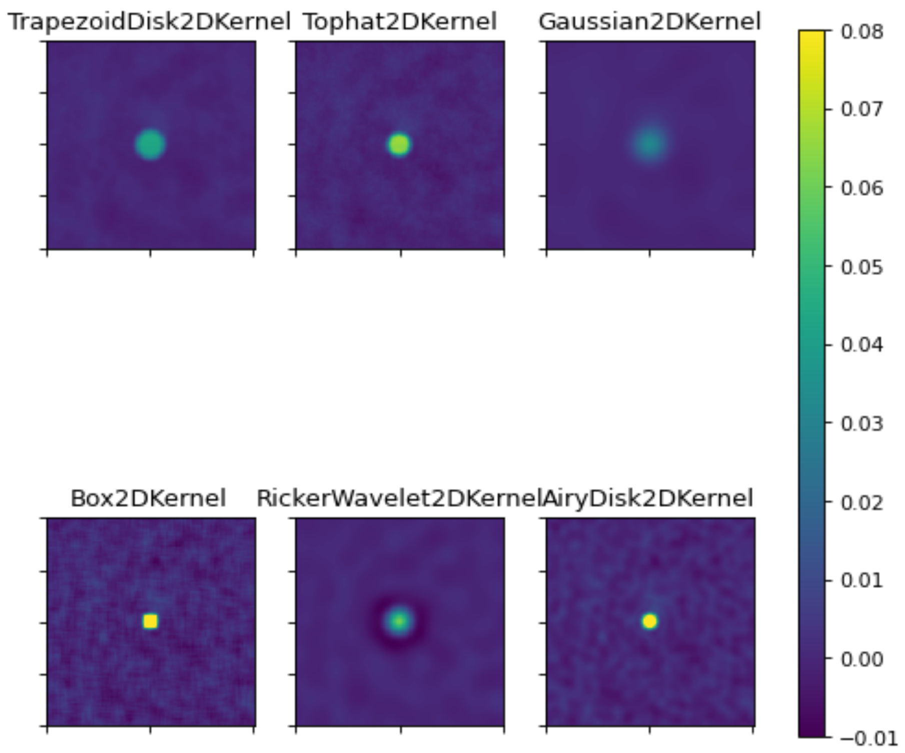
This is what the original image looks like:

(png, svg, pdf)



The following plot illustrates the differences between several 2D kernels applied to the simulated data. Note that it has a slightly different color scale compared to the original image.

(png, svg, pdf)



The Gaussian kernel has better smoothing properties compared to the Box and the Top Hat. The Box filter is not isotropic and can produce artifacts (the source appears rectangular). The Ricker Wavelet filter removes noise and slowly varying structures (i.e., background), but produces a negative ring around the source. The best choice for the filter strongly depends on the application.

Available Kernels

`AiryDisk2DKernel`(radius, **kwargs)	2D Airy disk kernel.
`Box1DKernel`(width, **kwargs)	1D Box filter kernel.
`Box2DKernel`(width, **kwargs)	2D Box filter kernel.
`CustomKernel`(array)	Create filter kernel from list or array.
`Gaussian1DKernel`(stddev, **kwargs)	1D Gaussian filter kernel.
`Gaussian2DKernel`(x_stddev[, y_stddev, theta])	2D Gaussian filter kernel.
`RickerWavelet1DKernel`(width, **kwargs)	1D Ricker wavelet filter kernel (sometimes known as a “Mexican Hat” kernel).
`RickerWavelet2DKernel`(width, **kwargs)	2D Ricker wavelet filter kernel (sometimes known as a “Mexican Hat” kernel).
`Model1DKernel`(model, **kwargs)	Create kernel from 1D model.
`Model2DKernel`(model, **kwargs)	Create kernel from 2D model.
`Ring2DKernel`(radius_in, width, **kwargs)	2D Ring filter kernel.

`Tophat2DKernel`(radius, **kwargs)	2D Tophat filter kernel.
`Trapezoid1DKernel`(width[, slope])	1D trapezoid kernel.
`TrapezoidDisk2DKernel`(radius[, slope])	2D trapezoid kernel.

Kernel Arithmetics

Addition and Subtraction

As convolution is a linear operation, kernels can be added or subtracted from each other. They can also be multiplied with some number.

Examples

One basic example of subtracting kernels would be the definition of a Difference of Gaussian filter:

```
>>> from astropy.convolution import Gaussian1DKernel
>>> gauss_1 = Gaussian1DKernel(10)
>>> gauss_2 = Gaussian1DKernel(16)
>>> DoG = gauss_2 - gauss_1
```

Another application is to convolve faked data with an instrument response function model. For example, if the response function can be described by the weighted sum of two Gaussians:

```
>>> gauss_1 = Gaussian1DKernel(10)
>>> gauss_2 = Gaussian1DKernel(16)
>>> SoG = 4 * gauss_1 + gauss_2
```

Most times it will be necessary to normalize the resulting kernel by calling explicitly:

```
>>> SoG.normalize()
```

Convolution

Furthermore, two kernels can be convolved with each other, which is useful when data is filtered with two different kinds of kernels or to create a new, special kernel.

Examples

To convolve two kernels with each other:

```
>>> from astropy.convolution import Gaussian1DKernel, convolve
>>> gauss_1 = Gaussian1DKernel(10)
>>> gauss_2 = Gaussian1DKernel(16)
>>> broad_gaussian = convolve(gauss_2,  gauss_1)
```

Or in case of multistage smoothing:

```
>>> import numpy as np
>>> from astropy.modeling.models import Lorentz1D
>>> from astropy.convolution import convolve, Gaussian1DKernel,
Box1DKernel
>>> lorentz = Lorentz1D(1, 0, 1)
>>> x = np.linspace(-5, 5, 100)
>>> data_1D = lorentz(x) + 0.1 * (np.random.rand(100) - 0.5)
```

```
>>> gauss = Gaussian1DKernel(3)
>>> box = Box1DKernel(5)
>>> smoothed_gauss = convolve(data_1D, gauss)
>>> smoothed_gauss_box = convolve(smoothed_gauss, box)
```

You would rather do the following:

```
>>> gauss = Gaussian1DKernel(3)
>>> box = Box1DKernel(5)
>>> smoothed_gauss_box = convolve(data_1D, convolve(box, gauss))
```

Which, in most cases, will also be faster than the first method because only one convolution with the often times larger data array will be necessary.

Discretization

To obtain the kernel array for discrete convolution, the kernel’s response function is evaluated on a grid with **`discretize_model()`**. For the discretization step the following modes are available:

- Mode `'center'` (default) evaluates the response function on the grid by taking the value at the center of the bin.

  ```
  >>> from astropy.convolution import Gaussian1DKernel
  >>> gauss_center = Gaussian1DKernel(3, mode='center')
  ```

- Mode `'linear_interp'` takes the values at the corners of the bin and linearly interpolates the value at the center:

```
>>> gauss_interp = Gaussian1DKernel(3, mode='linear_interp')
```

- Mode `'oversample'` evaluates the response function by taking the mean on an oversampled grid. The oversample factor can be specified with the `factor` argument. If the oversample factor is too large, the evaluation becomes slow.

```
>>> gauss_oversample = Gaussian1DKernel(3, mode='oversample',
factor=10)
```

- Mode `'integrate'` integrates the function over the pixel using `scipy.integrate.quad` and `scipy.integrate.dblquad`. This mode is very slow and is only recommended when the highest accuracy is required.

```
>>> gauss_integrate = Gaussian1DKernel(3, mode='integrate')
```

Especially in the range where the kernel width is in order of only a few pixels, it can be advantageous to use the mode `oversample` or `integrate` to conserve the integral on a subpixel scale.

Normalization

The kernel models are normalized per default (i.e., \(\int_{-\infty}^{\infty} f(x) dx = 1\)). But because of the limited kernel array size, the normalization for kernels with an infinite response can differ from one. The value of this deviation is stored in the kernel's `truncation` attribute.

The normalization can also differ from one, especially for small kernels, due to the discretization step. This can be partly controlled by the `mode` argument, when initializing the kernel. (See also **`discretize_model()`**.) Setting the `mode` to `'oversample'` allows us to conserve the normalization even on the subpixel scale.

The kernel arrays can be renormalized explicitly by calling either the `normalize()` method or by setting the `normalize_kernel` argument in the **`convolve()`** and **`convolve_fft()`** functions. The latter method leaves the kernel itself unchanged but works with an internal normalized version of the kernel.

Note that for **`RickerWavelet1DKernel`** and **`RickerWavelet2DKernel`** there is \(\int_{-\infty}^{\infty} f(x) dx = 0\). To define a proper normalization, both filters are derived from a normalized Gaussian function.

Convolving with Unnormalized Kernels

There are some tasks, such as source finding, where you want to apply a filter with a kernel that is not normalized.

For data that are well-behaved (contain no missing or infinite values), this can be done in one step:

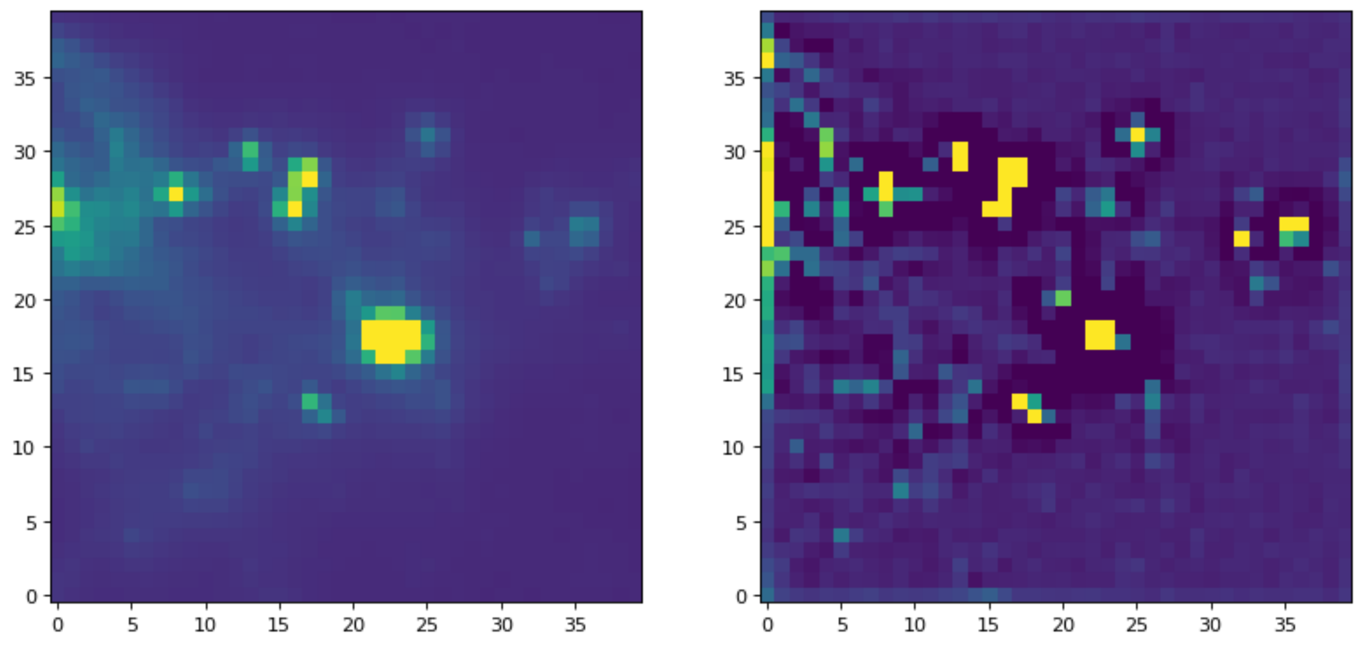
```
convolve(image, kernel)
```

Examples

For an example of applying a filter with a kernel that is not normalized, we can try to run a commonly used peak enhancing kernel:

```
import numpy as np
import matplotlib.pyplot as plt

from astropy.io import fits
from astropy.utils.data import get_pkg_data_filename
from astropy.convolution import CustomKernel
from scipy.signal import convolve as scipy_convolve
from astropy.convolution import convolve, convolve_fft


# Load the data from data.astropy.org
filename = get_pkg_data_filename('galactic_center/gc_msx_e.fits')
hdu = fits.open(filename)[0]

# Scale the file to have reasonable numbers
# (this is mostly so that colorbars don't have too many digits)
# Also, we crop it so you can see individual pixels
img = hdu.data[50:90, 60:100] * 1e5

kernel = CustomKernel([[-1,-1,-1], [-1, 8, -1], [-1,-1,-1]])

astropy_conv = convolve(img, kernel, normalize_kernel=False,
nan_treatment='fill')
#astropy_conv_fft = convolve_fft(img, kernel, normalize_kernel=False,
nan_treatment='fill')

plt.figure(1, figsize=(12, 12)).clf()
ax1 = plt.subplot(1, 2, 1)
im = ax1.imshow(img, vmin=-6., vmax=5.e1, origin='lower',
                interpolation='nearest', cmap='viridis')

ax2 = plt.subplot(1, 2, 2)
im = ax2.imshow(astropy_conv, vmin=-6., vmax=5.e1, origin='lower',
                interpolation='nearest', cmap='viridis')
```

(png, svg, pdf)

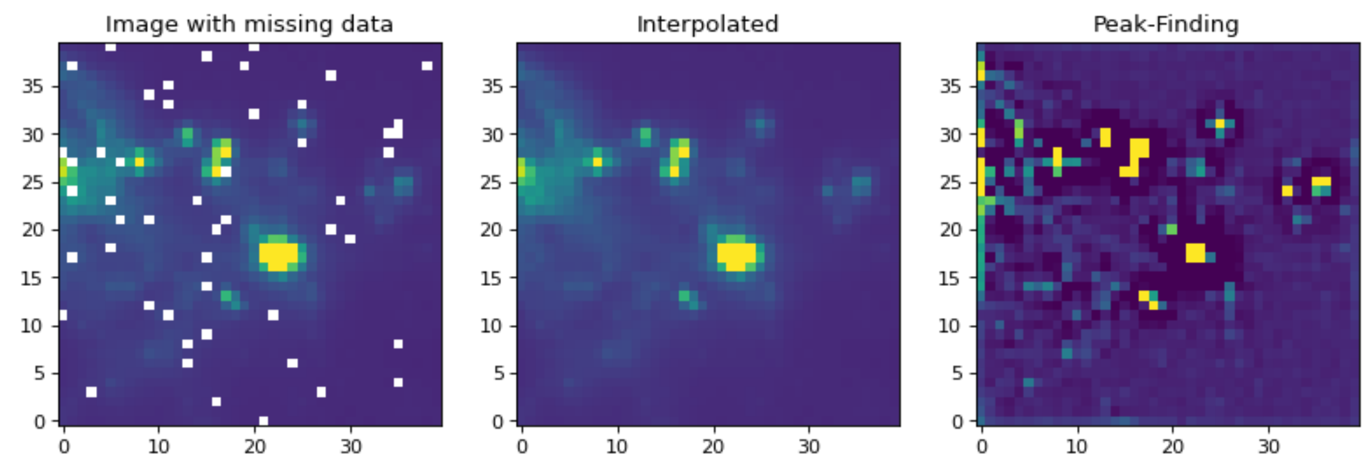


If you have an image with missing values (NaNs), you have to replace them with real values first. Often, the best way to do this is to replace the NaN values with interpolated values. In the example below, we use a Gaussian kernel with a size similar to that of our peak-finding kernel to replace the bad data before applying the peak-finding kernel.

```
from astropy.convolution import Gaussian2DKernel,
interpolate_replace_nans

# Select a random set of pixels that were affected by some sort of
artifact
# and replaced with NaNs (e.g., cosmic-ray-affected pixels)
np.random.seed(42)
yinds, xinds = np.indices(img.shape)
img[np.random.choice(yinds.flat, 50), np.random.choice(xinds.flat,
50)] = np.nan

# We smooth with a Gaussian kernel with x_stddev=1 (and y_stddev=1)
# It is a 9x9 array
kernel = Gaussian2DKernel(x_stddev=1)

# interpolate away the NaNs
```

```
reconstructed_image = interpolate_replace_nans(img, kernel)


# apply peak-finding
kernel = CustomKernel([[-1,-1,-1], [-1, 8, -1], [-1,-1,-1]])

# Use the peak-finding kernel
# We have to turn off kernel normalization and set nan_treatment to
"fill"
# here because `nan_treatment='interpolate'` is incompatible with
non-
# normalized kernels
peaked_image = convolve(reconstructed_image, kernel,
                        normalize_kernel=False,
                        nan_treatment='fill')

plt.figure(1, figsize=(12, 12)).clf()
ax1 = plt.subplot(1, 3, 1)
ax1.set_title("Image with missing data")
im = ax1.imshow(img, vmin=-6., vmax=5.e1, origin='lower',
                interpolation='nearest', cmap='viridis')

ax2 = plt.subplot(1, 3, 2)
ax2.set_title("Interpolated")
im = ax2.imshow(reconstructed_image, vmin=-6., vmax=5.e1,
origin='lower',
                interpolation='nearest', cmap='viridis')

ax3 = plt.subplot(1, 3, 3)
ax3.set_title("Peak-Finding")
im = ax3.imshow(peaked_image, vmin=-6., vmax=5.e1, origin='lower',
                interpolation='nearest', cmap='viridis')
```

(png, svg, pdf)



Performance Tips

The `convolve()` function is best suited to small kernels, and can become very slow for larger kernels. In this case, consider using `convolve_fft()` (though note that this function uses more memory).

Reference/API

astropy.convolution Package

Functions

`convolve`(array, kernel[, boundary, …])	Convolve an array with a kernel.
`convolve_fft`(array, kernel[, boundary, …])	Convolve an ndarray with an nd-kernel.
`convolve_models`(model, kernel[, mode])	Convolve two models using `convolve_fft`.
`discretize_model`(model, x_range[, y_range, …])	Function to evaluate analytical model functions on a grid.
`interpolate_replace_nans`(array, kernel[, …])	Given a data set containing NaNs, replace the NaNs by interpolating from neighboring data points with a given kernel.
`kernel_arithmetics`(kernel, value, operation)	Add, subtract or multiply two kernels.

Classes

`AiryDisk2DKernel`(radius, **kwargs)	2D Airy disk kernel.
`Box1DKernel`(width, **kwargs)	1D Box filter kernel.
`Box2DKernel`(width, **kwargs)	2D Box filter kernel.
`CustomKernel`(array)	Create filter kernel from list or array.
`Gaussian1DKernel`(stddev, **kwargs)	1D Gaussian filter kernel.
`Gaussian2DKernel`(x_stddev[, y_stddev, theta])	2D Gaussian filter kernel.
`Kernel`(array)	Convolution kernel base class.
`Kernel1D`([model, x_size, array])	Base class for 1D filter kernels.
`Kernel2D`([model, x_size, y_size, array])	Base class for 2D filter kernels.
`Model1DKernel`(model, **kwargs)	Create kernel from 1D model.
`Model2DKernel`(model, **kwargs)	Create kernel from 2D model.

`Moffat2DKernel`(gamma, alpha, **kwargs)	2D Moffat kernel.
`RickerWavelet1DKernel`(width, **kwargs)	1D Ricker wavelet filter kernel (sometimes known as a “Mexican Hat” kernel).
`RickerWavelet2DKernel`(width, **kwargs)	2D Ricker wavelet filter kernel (sometimes known as a “Mexican Hat” kernel).
`Ring2DKernel`(radius_in, width, **kwargs)	2D Ring filter kernel.
`Tophat2DKernel`(radius, **kwargs)	2D Tophat filter kernel.
`Trapezoid1DKernel`(width[, slope])	1D trapezoid kernel.
`TrapezoidDisk2DKernel`(radius[, slope])	2D trapezoid kernel.

Data Visualization (`astropy.visualization`)

Introduction

`astropy.visualization` provides functionality that can be helpful when visualizing data. This includes a framework for plotting Astronomical images with coordinates with Matplotlib (previously the standalone **wcsaxes** package), functionality related to image normalization (including both scaling and stretching), smart histogram plotting, RGB color image creation from separate images, and custom plotting styles for Matplotlib.

Using `astropy.visualization`

Plotting Astropy objects in Matplotlib

Plotting quantities

`Quantity` objects can be conveniently plotted using matplotlib. This feature needs to be explicitly turned on:

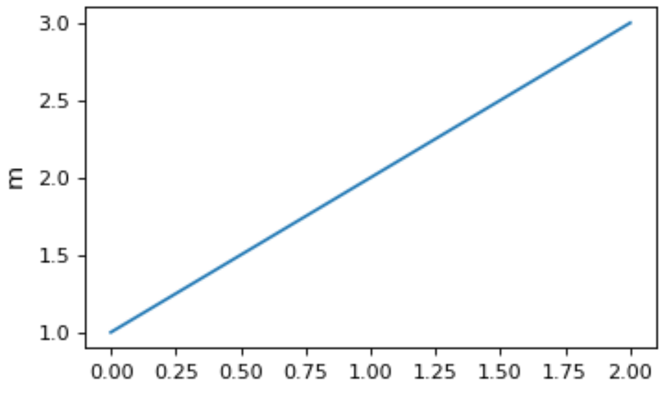
```
>>> from astropy.visualization import quantity_support
>>> quantity_support()
<astropy.visualization.units.MplQuantityConverter ...>
```

Then `Quantity` objects can be passed to matplotlib plotting functions. The axis labels are automatically labeled with the unit of the quantity:

```
from astropy import units as u
from astropy.visualization import quantity_support
quantity_support()
from matplotlib import pyplot as plt
```

```
plt.figure(figsize=(5,3))
plt.plot([1, 2, 3] * u.m)
```

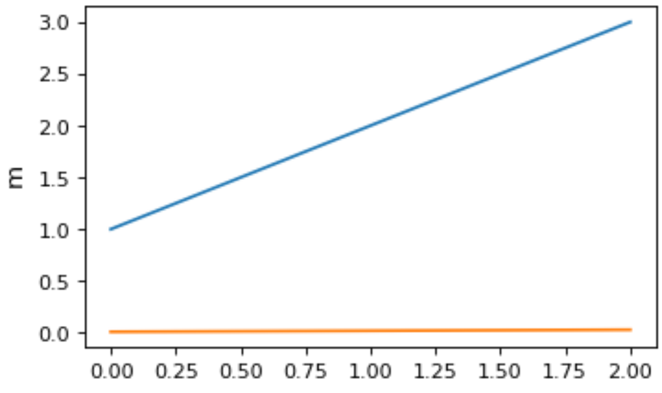
(png, svg, pdf)



Quantities are automatically converted to the first unit set on a particular axis, so in the following, the y-axis remains in `m` even though the second line is given in `cm`:

```
plt.plot([1, 2, 3] * u.cm)
```

(png, svg, pdf)



Plotting a quantity with an incompatible unit will raise an exception. For example, calling `plt.plot([1, 2, 3] * u.kg)` (mass unit) to overplot on the plot above that is displaying length units.

To make sure unit support is turned off afterward, you can use **`quantity_support`** with a `with` statement:

```
with quantity_support():
    plt.plot([1, 2, 3] * u.m)
```

Plotting times

Matplotlib natively provides a mechanism for plotting dates and times on one or both of the axes, as described in Date tick labels. To make use of this, you can use the `plot_date` attribute of **`Time`** to get values in the time system used by Matplotlib.

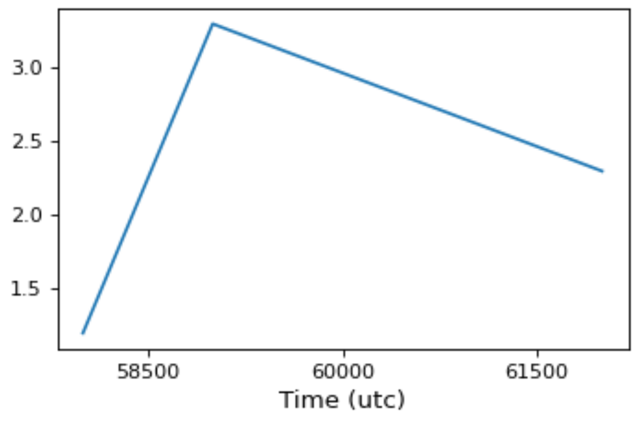
However, in many cases, you will probably want to have more control over the precise scale and format to use for the tick labels, in which case you can make use of the **`time_support`** function. This feature needs to be explicitly turned on:

```
>>> from astropy.visualization import time_support
>>> time_support()
<astropy.visualization.units.MplTimeConverter ...>
```

Once this is enabled, **`Time`** objects can be passed to matplotlib plotting functions. The axis labels are then automatically labeled with times formatted using the **`Time`** class:

```
from matplotlib import pyplot as plt
from astropy.time import Time
from astropy.visualization import time_support

time_support()

plt.figure(figsize=(5,3))
plt.plot(Time([58000, 59000, 62000], format='mjd'), [1.2, 3.3, 2.3])
```

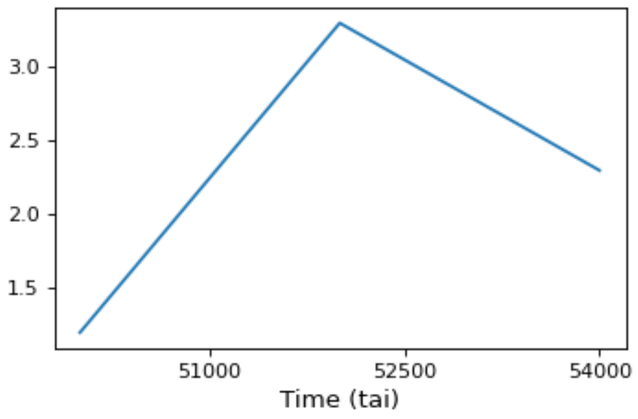
(png, svg, pdf)



By default, the format and scale used for the plots is taken from the first time that Matplotlib encounters for a particular Axes instance. The format and scale can also be explicitly controlled by passing arguments to `time_support`:

```
time_support(format='mjd', scale='tai')
plt.figure(figsize=(5,3))
plt.plot(Time([50000, 52000, 54000], format='mjd'), [1.2, 3.3, 2.3])
```

(png, svg, pdf)



To make sure support for plotting times is turned off afterward, you can use **time_support** as a context manager:

```
with time_support(format='mjd', scale='tai'):
    plt.figure(figsize=(5,3))
    plt.plot(Time([50000, 52000, 54000], format='mjd'))
```

Making plots with world coordinates (WCSAxes)

WCSAxes is a framework for making plots of Astronomical data in Matplotlib. It was previously distributed as a standalone package, but is now included in astropy.visualization.

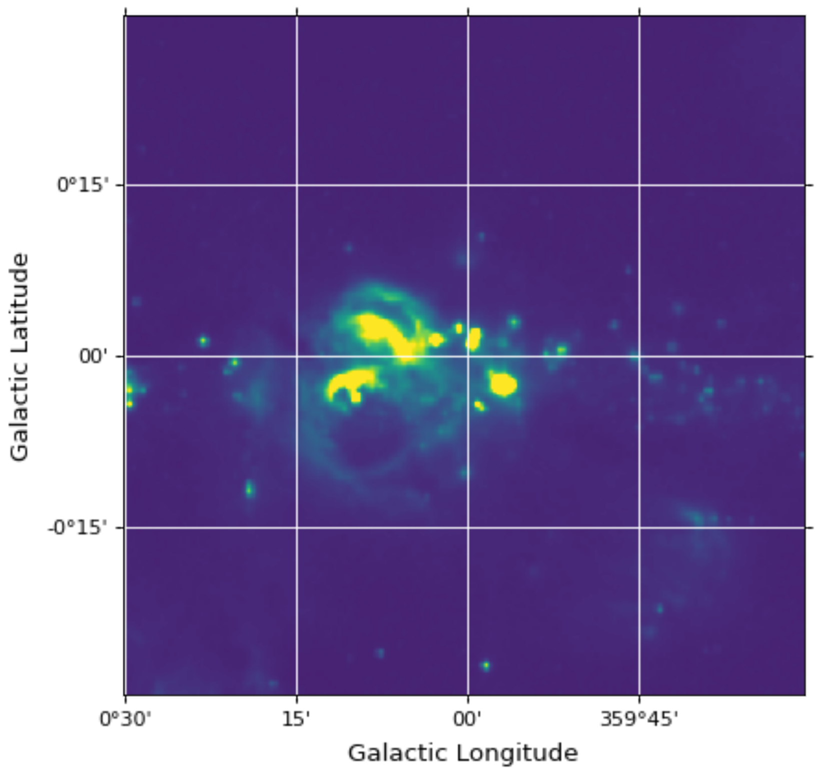
Getting started

The following is a very simple example of plotting an image with the WCSAxes package:

```
import matplotlib.pyplot as plt

from astropy.wcs import WCS
from astropy.io import fits
from astropy.utils.data import get_pkg_data_filename

filename = get_pkg_data_filename('galactic_center/gc_msx_e.fits')
```

```
hdu = fits.open(filename)[0]
wcs = WCS(hdu.header)

plt.subplot(projection=wcs)
plt.imshow(hdu.data, vmin=-2.e-5, vmax=2.e-4, origin='lower')
plt.grid(color='white', ls='solid')
plt.xlabel('Galactic Longitude')
plt.ylabel('Galactic Latitude')
```

(png, svg, pdf)



This example uses the **`matplotlib.pyplot`** interface to Matplotlib, but WCSAxes can be used with any of the other ways of using Matplotlib (some examples of which are given in Initializing axes with world coordinates). For example, using the partially object-oriented interface, you can do:

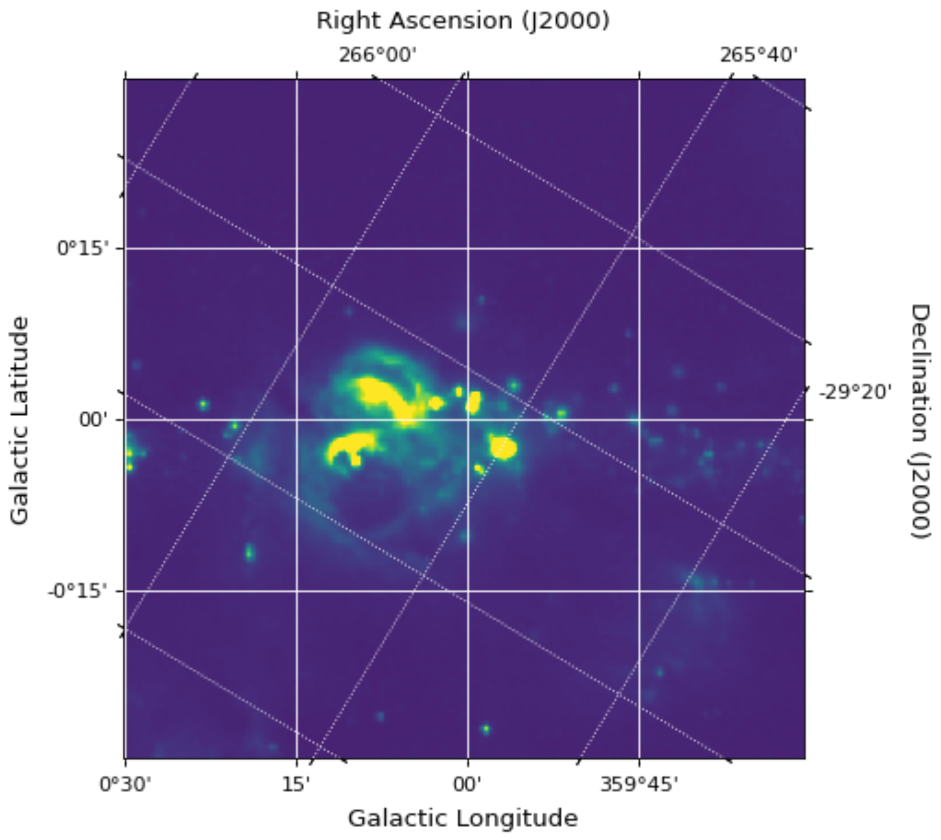
```
ax = plt.subplot(projection=wcs)
ax.imshow(hdu.data, vmin=-2.e-5, vmax=2.e-4, origin='lower')
ax.grid(color='white', ls='solid')
ax.set_xlabel('Galactic Longitude')
ax.set_ylabel('Galactic Latitude')
```

However, the axes object is needed to access some of the more advanced functionality of WCSAxes. An example of this usage is:

```
ax = plt.subplot(projection=wcs, label='overlays')
```

```
ax.imshow(hdu.data, vmin=-2.e-5, vmax=2.e-4, origin='lower')

ax.coords.grid(True, color='white', ls='solid')
ax.coords[0].set_axislabel('Galactic Longitude')
ax.coords[1].set_axislabel('Galactic Latitude')

overlay = ax.get_coords_overlay('fk5')
overlay.grid(color='white', ls='dotted')
overlay[0].set_axislabel('Right Ascension (J2000)')
overlay[1].set_axislabel('Declination (J2000)')
```

(png, svg, pdf)



In the rest of this documentation we will assume that you have kept a reference to the axes object, which we will refer to as `ax`. However, we also note when something can be done directly with the pyplot interface.

WCSAxes supports a number of advanced plotting options, including the ability to control which axes to show labels on for which coordinates, overlaying contours from data with different coordinate systems, overlaying grids for different coordinate systems, dealing with plotting slices from data with more dimensions than the plot, and defining custom (non-rectangular) frames.

Using WCSAxes

Initializing axes with world coordinates

Basic initialization

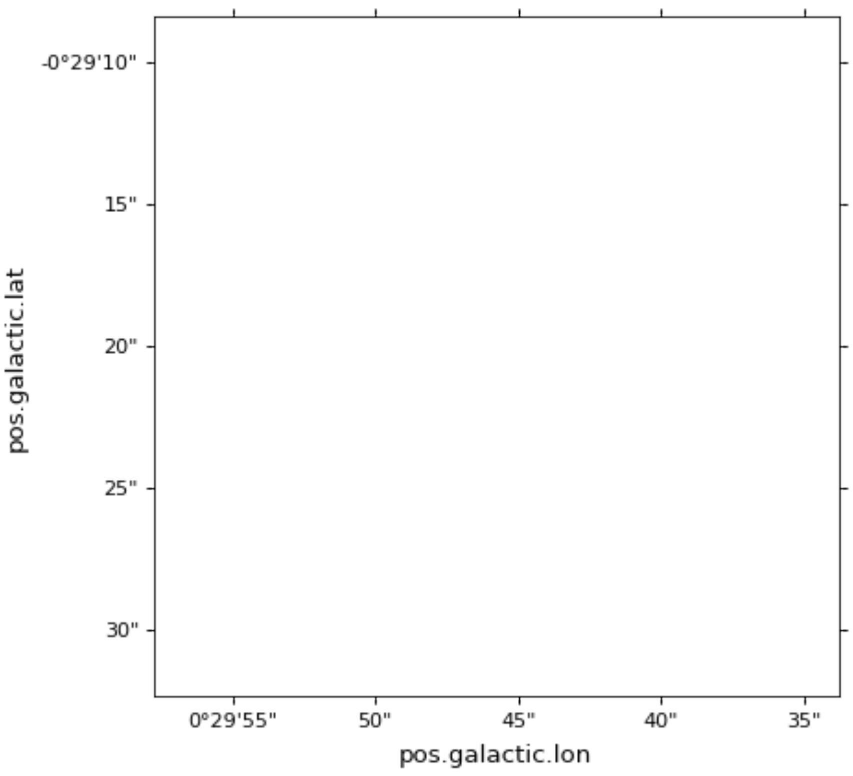
To make a plot using **WCSAxes**, we first read in the data using astropy.io.fits and parse the WCS information. In this example, we will use an example FITS file from the http://data.astropy.org server (the **get_pkg_data_filename()** function downloads the file and returns a filename):

```
from astropy.wcs import WCS
from astropy.io import fits
from astropy.utils.data import get_pkg_data_filename

filename = get_pkg_data_filename('galactic_center/gc_msx_e.fits')

hdu = fits.open(filename)[0]
wcs = WCS(hdu.header)
```

We then create a figure using Matplotlib and create the axes using the **WCS** object created above. The following example shows how to do this with the Matplotlib ‘pyplot’ interface, keeping a reference to the axes object:

```
import matplotlib.pyplot as plt
ax = plt.subplot(projection=wcs)
```

(png, svg, pdf)

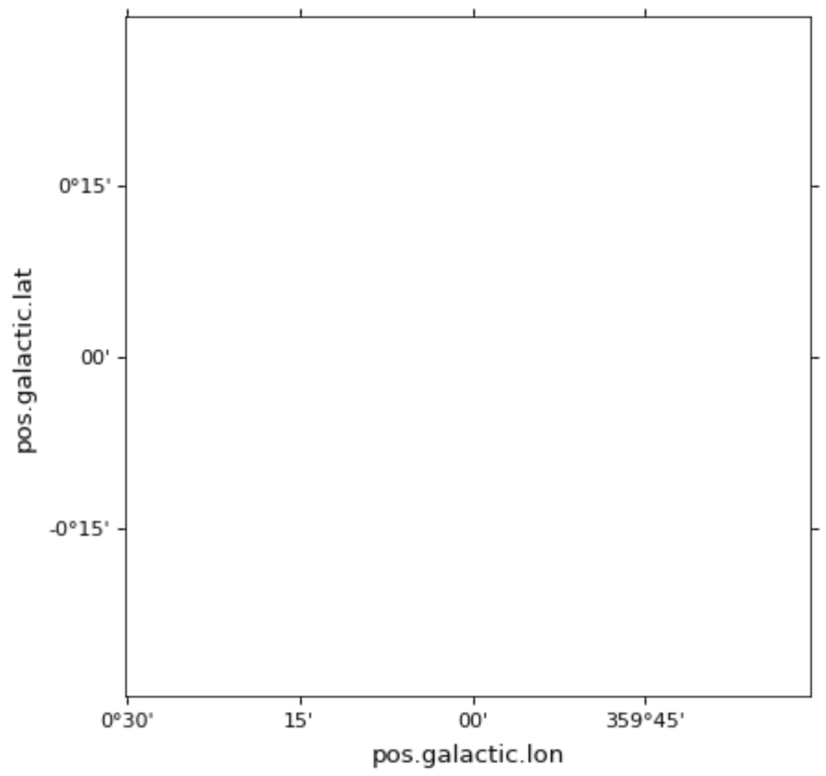


The `ax` object created is an instance of the **`WCSAxes`** class. Note that if no WCS transformation is specified, the transformation will default to identity, meaning that the world coordinates will match the pixel coordinates.

The field of view shown is, as for standard matplotlib axes, 0 to 1 in both directions, in pixel coordinates. As soon as you show an image (see Plotting images and contours), the limits will be adjusted, but if you want you can also adjust the limits manually. Adjusting the limits is done using the same functions/methods as for a normal Matplotlib plot:

```
ax.set_xlim(-0.5, hdu.data.shape[1] - 0.5)
ax.set_ylim(-0.5, hdu.data.shape[0] - 0.5)
```

(png, svg, pdf)



Note

If you use the pyplot interface, you can also replace `ax.set_xlim` and `ax.set_ylim` by `plt.xlim` and `plt.ylim`.

Alternative methods

As in Matplotlib, there are in fact several ways you can initialize the **`WCSAxes`**.

As shown above, the simplest way is to make use of the **`WCS`** class and pass this to `plt.subplot`. If you normally use the (partially) object-oriented interface of Matplotlib, you can also do:

```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, projection=wcs)
```

Note that this also works with **`add_axes()`** and **`axes()`**, e.g.:

```
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8], projection=wcs)
```

or:

```
plt.axes([0.1, 0.1, 0.8, 0.8], projection=wcs)
```

Any additional arguments passed to **`add_subplot()`**, **`add_axes()`**, **`subplot()`**, or **`axes()`**, such as `slices` or `frame_class`, will be passed on to the `WCSAxes` class.

Directly initializing WCSAxes

As an alternative to the above methods of initializing `WCSAxes`, you can also instantiate `WCSAxes` directly and add it to the figure:

```
from astropy.wcs import WCS
from astropy.visualization.wcsaxes import WCSAxes
import matplotlib.pyplot as plt

wcs = WCS(...)

fig = plt.figure()
ax = WCSAxes(fig, [0.1, 0.1, 0.8, 0.8], wcs=wcs)
fig.add_axes(ax)  # note that the axes have to be explicitly added to
the figure
```

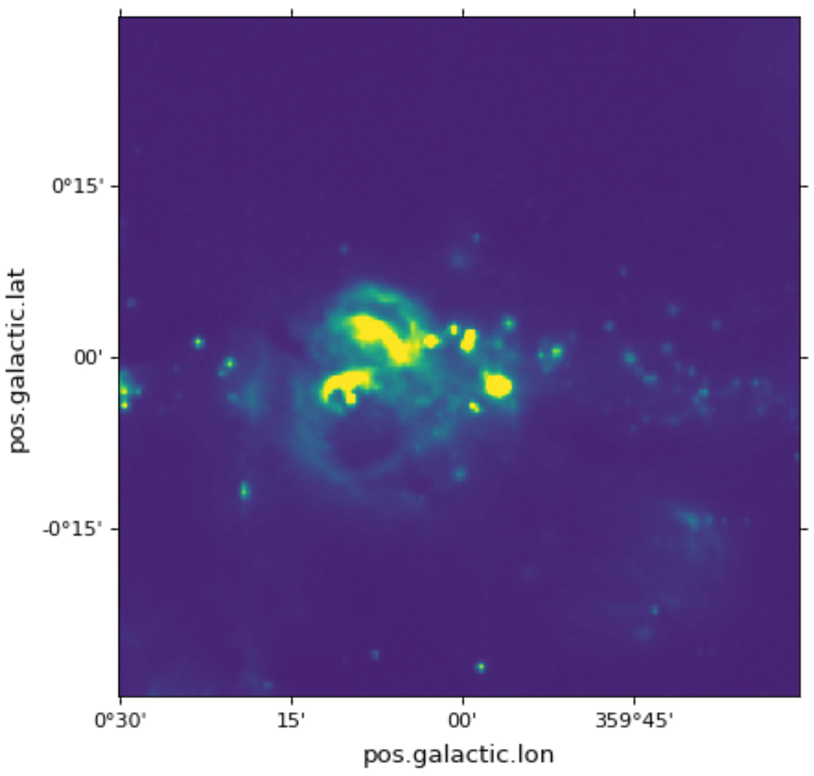
Plotting images and contours

For the example in the following page we start from the example introduced in Initializing axes with world coordinates.

Plotting images as bitmaps or contours should be done via the usual matplotlib methods such as `imshow()` or `contour()`. For example, continuing from the example in Initializing axes with world coordinates, you can do:

```
ax.imshow(hdu.data, vmin=-2.e-5, vmax=2.e-4, origin='lower')
```

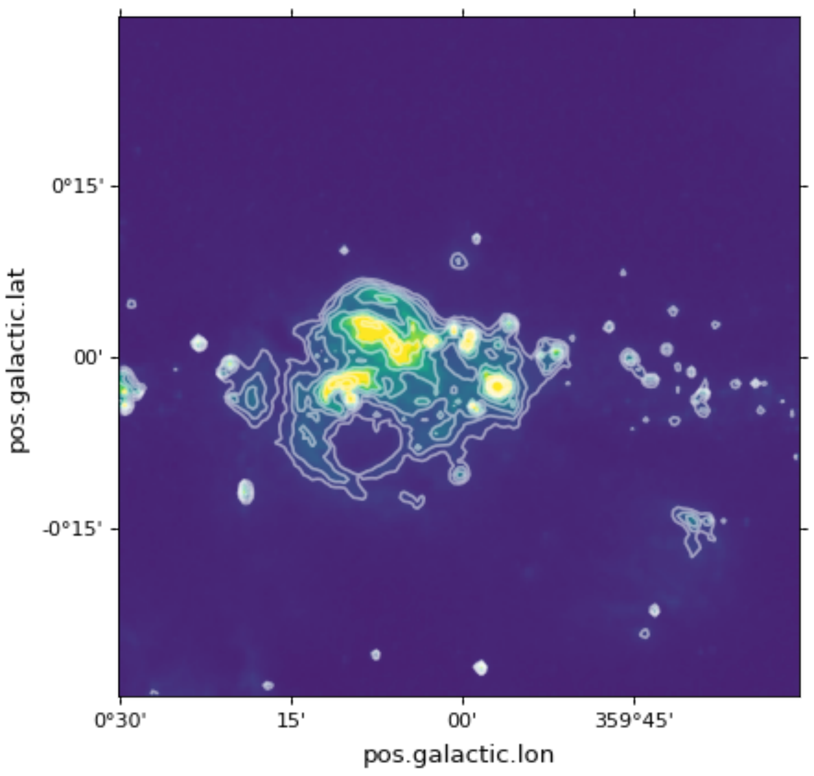
(png, svg, pdf)



and we can also add contours corresponding to the same image using:

```
import numpy as np
ax.contour(hdu.data, levels=np.logspace(-4.7, -3., 10),
colors='white', alpha=0.5)
```

(png, svg, pdf)



To show contours for an image in a different coordinate system, see Overplotting markers and artists.

Note

If you like using the pyplot interface, you can also call `plt.imshow` and `plt.contour` instead of `ax.imshow` and `ax.contour`.

Ticks, tick labels, and grid lines

For the example in the following page we start from the example introduced in Initializing axes with world coordinates.

Coordinate objects

While for many images, the coordinate axes are aligned with the pixel axes, this is not always the case, especially if there is any rotation in the world coordinate system, or in coordinate systems with high curvature, where the coupling between x- and y-axis to actual coordinates become less well-defined.

Therefore rather than referring to `x` and `y` ticks as Matplotlib does, we use specialized objects to access the coordinates. The coordinates used in the plot can be accessed using the `coords` attribute of the axes. As a reminder, if you use the pyplot interface, you can grab a reference to the axes when creating a subplot:

```
ax = plt.subplot()
```

or you can call `plt.gca()` at any time to get the current active axes:

```
ax = plt.gca()
```

If you use the object-oriented interface to Matplotlib, you should already have a reference to the axes.

Once you have an axes object, the coordinates can either be accessed by index:

```
lon = ax.coords[0]
lat = ax.coords[1]
```

or, in the case of common coordinate systems, by their name:

```
lon = ax.coords['glon']
lat = ax.coords['glat']
```

In this example, the image is in Galactic coordinates, so the coordinates are called `glon` and `glat`. For an image in equatorial coordinates, you would use `ra` and `dec`. The names are only available for specific celestial coordinate systems - for all other systems, you should use the index of the coordinate (`0` or `1`).

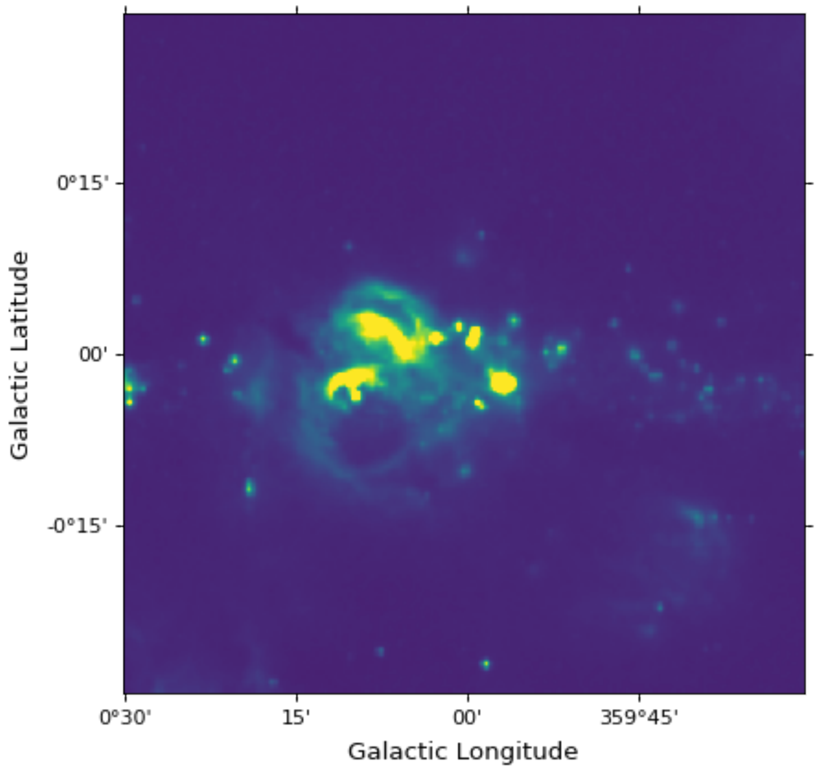
Each coordinate is an instance of the **`CoordinateHelper`** class, which can be used to control the appearance of the ticks, tick labels, grid lines, and axis labels associated with that coordinate.

Axis labels

Axis labels can be added using the **`set_axislabel()`** method:

```
lon.set_axislabel('Galactic Longitude')
lat.set_axislabel('Galactic Latitude')
```

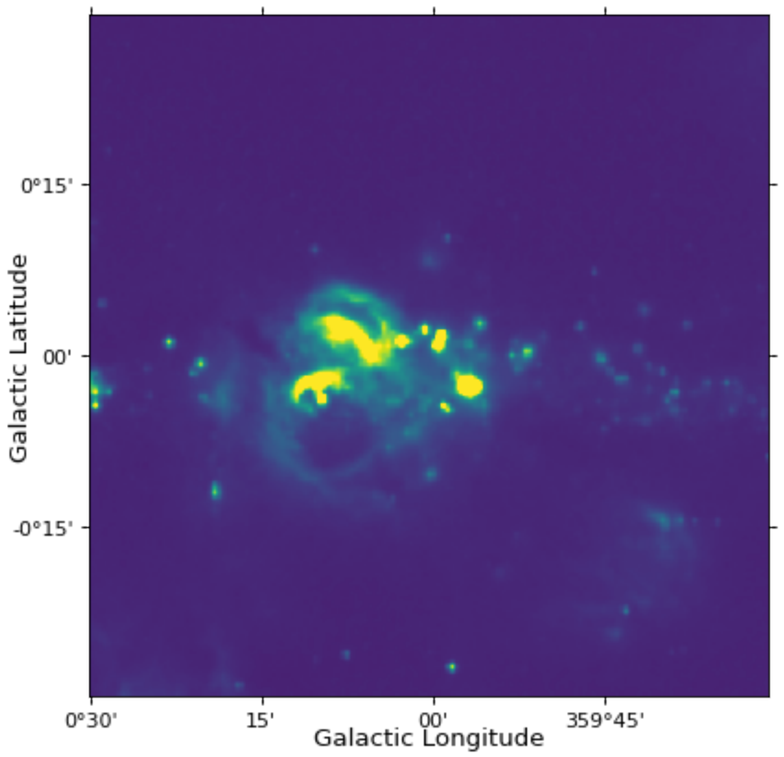
(png, svg, pdf)



The padding of the axis label with respect to the axes can also be adjusted by using the `minpad` option. The default value for `minpad` is 1 and is in terms of the font size of the axis label text. Negative values are also allowed.

```
lon.set_axislabel('Galactic Longitude', minpad=0.3)
lat.set_axislabel('Galactic Latitude', minpad=-0.4)
```

(png, svg, pdf)



Note

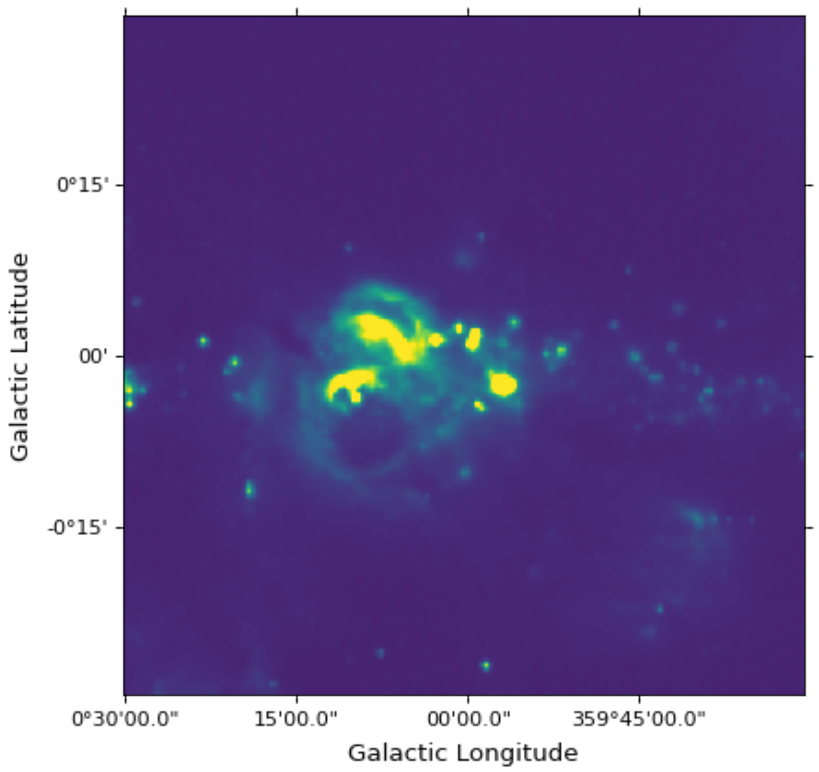
Note that, as shown in Getting started, it is also possible to use the normal `plt.xlabel` or `ax.set_xlabel` notation to set the axis labels in the case where they do appear on the x and y axis.

Tick label format

The format of the tick labels can be specified with a string describing the format:

```
lon.set_major_formatter('dd:mm:ss.s')
lat.set_major_formatter('dd:mm')
```

(png, svg, pdf)



The syntax for the format string is the following:

format	result
`'dd'`	`'15d'`
`'dd:mm'`	`'15d24m'`
`'dd:mm:ss'`	`'15d23m32s'`
`'dd:mm:ss.s'`	`'15d23m32.0s'`
`'dd:mm:ss.ssss'`	`'15d23m32.0316s'`
`'hh'`	`'1h'`
`'hh:mm'`	`'1h02m'`
`'hh:mm:ss'`	`'1h01m34s'`
`'hh:mm:ss.s'`	`'1h01m34.1s'`
`'hh:mm:ss.ssss'`	`'1h01m34.1354s'`
`'d'`	`'15'`
`'d.d'`	`'15.4'`
`'d.dd'`	`'15.39'`
`'d.ddd'`	`'15.392'`
`'m'`	`'924'`
`'m.m'`	`'923.5'`

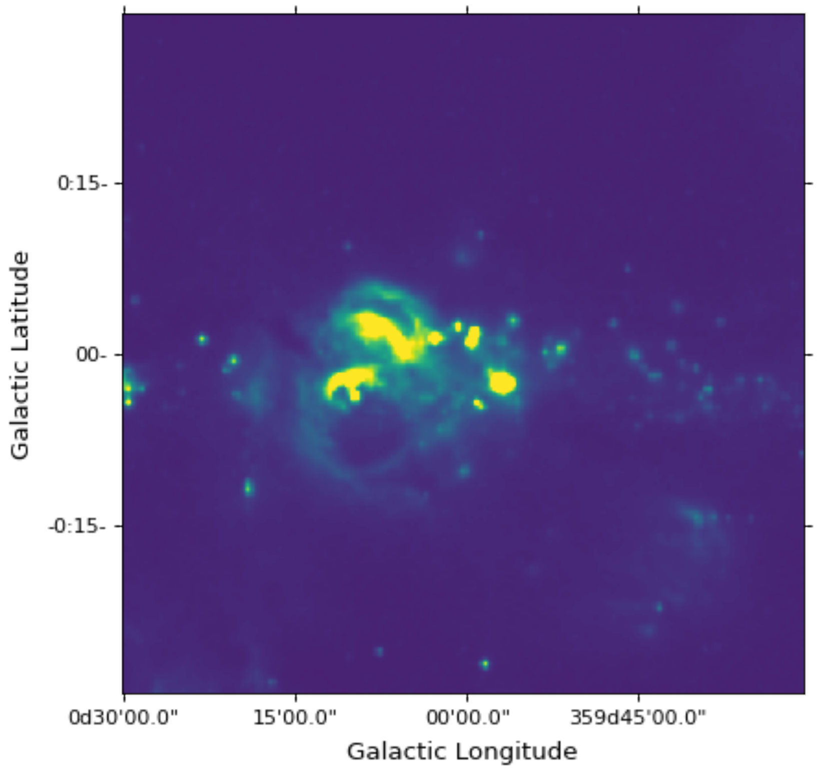
format	result
`'m.mm'`	`'923.53'`
`'s'`	`'55412'`
`'s.s'`	`'55412.0'`
`'s.ss'`	`'55412.03'`
`'x.xxxx'`	`'15.3922'`
`'%.2f'`	`'15.39'`
`'%.3f'`	`'15.392'`
`'%d'`	`'15'`

All the `h...`, `d...`, `m...`, and `s...` formats can be used for angular coordinate axes, while the `x...` format or valid Python formats (see String Formatting Operations) should be used for non-angular coordinate axes.

The separators for angular coordinate tick labels can also be set by specifying a string or a tuple.

```
lon.set_separator(('d', "'", '"'))
lat.set_separator(':-s')
```

(png, svg, pdf)



Tick/label spacing and properties

The spacing of ticks/tick labels should have a sensible default, but you may want to be able to manually specify the spacing. This can be done using the `set_ticks()` method. There are different options that can be used:

- Set the tick positions manually as an Astropy `Quantity`:

  ```
  from astropy import units as u
  lon.set_ticks([242.2, 242.3, 242.4] * u.degree)
  ```

- Set the spacing between ticks also as an Astropy `Quantity`:

  ```
  lon.set_ticks(spacing=5. * u.arcmin)
  ```

- Set the approximate number of ticks:

  ```
  lon.set_ticks(number=4)
  ```

In the case of angular axes, specifying the spacing as an Astropy `Quantity` avoids roundoff errors. The `set_ticks()` method can also be used to set the appearance (color and size) of the ticks, using the `color=` and `size=` options.

The `set_ticklabel()` method can be used to change settings for the tick labels, such as color, font, size, and so on:

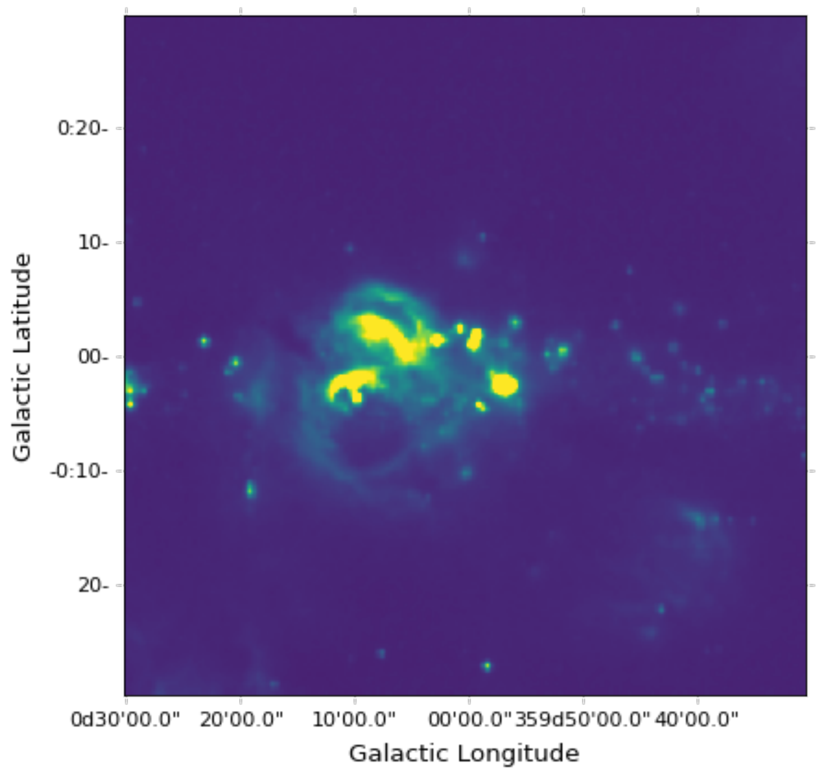
```
lon.set_ticklabel(color='red', size=12)
```

In addition, this method has an option `exclude_overlapping=True` to prevent overlapping tick labels from being displayed.

We can apply this to the previous example:

```
from astropy import units as u
lon.set_ticks(spacing=10 * u.arcmin, color='white')
lat.set_ticks(spacing=10 * u.arcmin, color='white')
lon.set_ticklabel(exclude_overlapping=True)
lat.set_ticklabel(exclude_overlapping=True)
```

(png, svg, pdf)

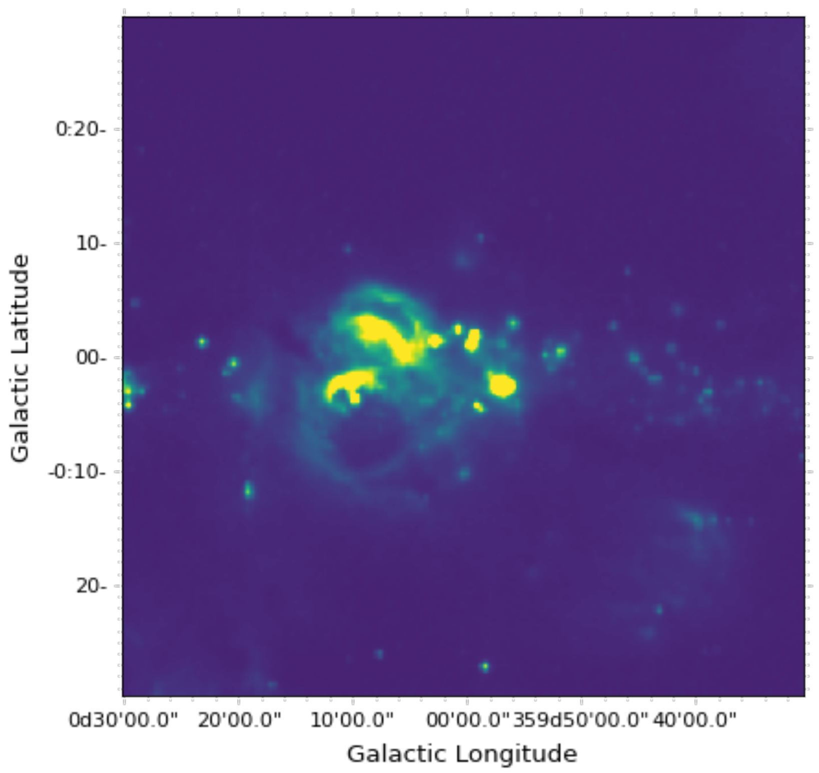


Minor ticks

WCSAxes does not display minor ticks by default but these can be shown by using the `display_minor_ticks()` method. The default frequency of minor ticks is 5 but this can also be specified.

```
lon.display_minor_ticks(True)
lat.display_minor_ticks(True)
lat.set_minor_frequency(10)
```

(png, svg, pdf)

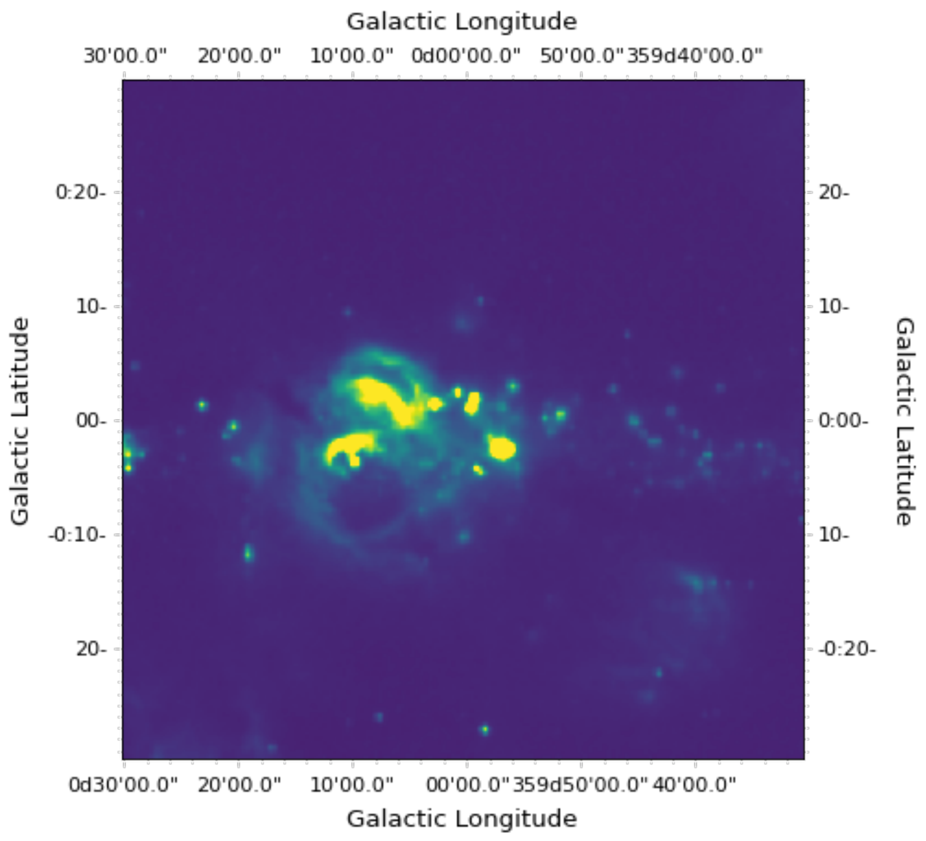


Tick, tick label, and axis label position

By default, the tick and axis labels for the first coordinate are shown on the x-axis, and the tick and axis labels for the second coordinate are shown on the y-axis. In addition, the ticks for both coordinates are shown on all axes. This can be customized using the `set_ticks_position()` and `set_ticklabel_position()` methods, which each take a string that can contain any or several of `l`, `b`, `r`, or `t` (indicating the ticks or tick labels should be shown on the left, bottom, right, or top axes respectively):

```
lon.set_ticks_position('bt')
lon.set_ticklabel_position('bt')
lon.set_axislabel_position('bt')
lat.set_ticks_position('lr')
lat.set_ticklabel_position('lr')
lat.set_axislabel_position('lr')
```

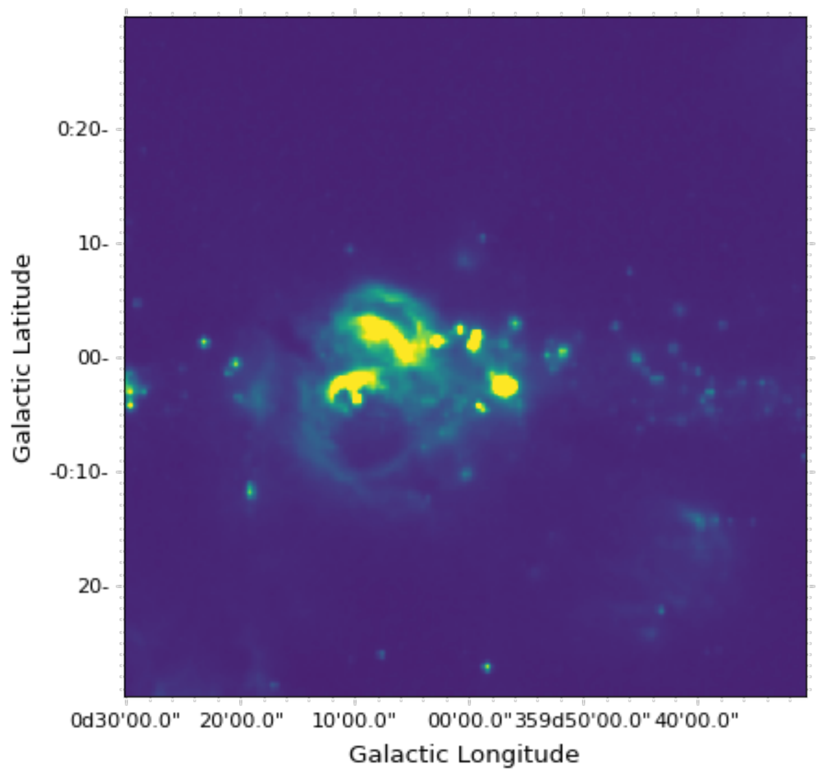
(png, svg, pdf)



We can set the defaults back using:

```
lon.set_ticks_position('all')
lon.set_ticklabel_position('b')
lon.set_axislabel_position('b')
lat.set_ticks_position('all')
lat.set_ticklabel_position('l')
lat.set_axislabel_position('l')
```

(png, svg, pdf)



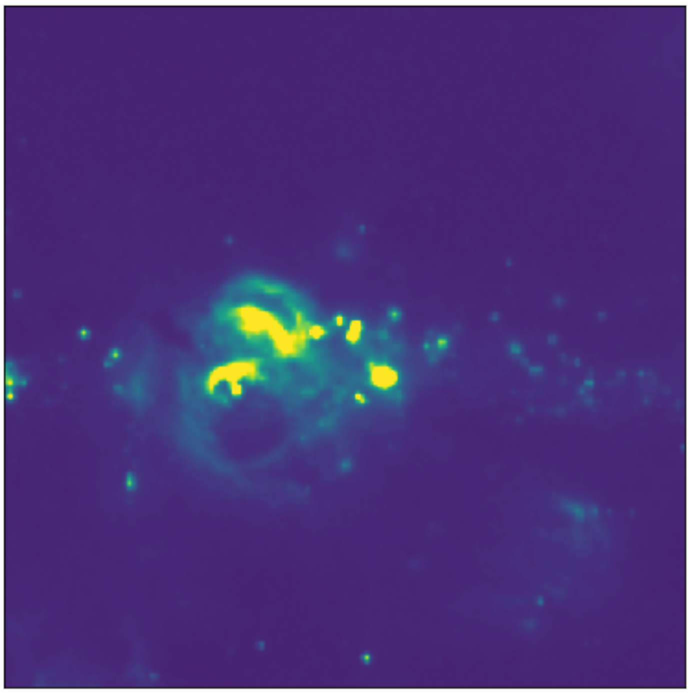
On plots with elliptical frames, three alternate tick positions are supported: `c` for the outer circular or elliptical border, `h` for the horizontal axis (which is usually the major axis of the ellipse), and `v` for the vertical axis (which is usually the minor axis of the ellipse).

Hiding ticks and tick labels

Sometimes it's desirable to hide ticks and tick labels. A common scenario is where WCSAxes is being used in a grid of subplots and the tick labels are redundant across rows or columns. Tick labels and ticks can be hidden with the **`set_ticklabel_visible()`** and **`set_ticks_visible()`** methods, respectively:

```
lon.set_ticks_visible(False)
lon.set_ticklabel_visible(False)
lat.set_ticks_visible(False)
lat.set_ticklabel_visible(False)
lon.set_axislabel('')
lat.set_axislabel('')
```

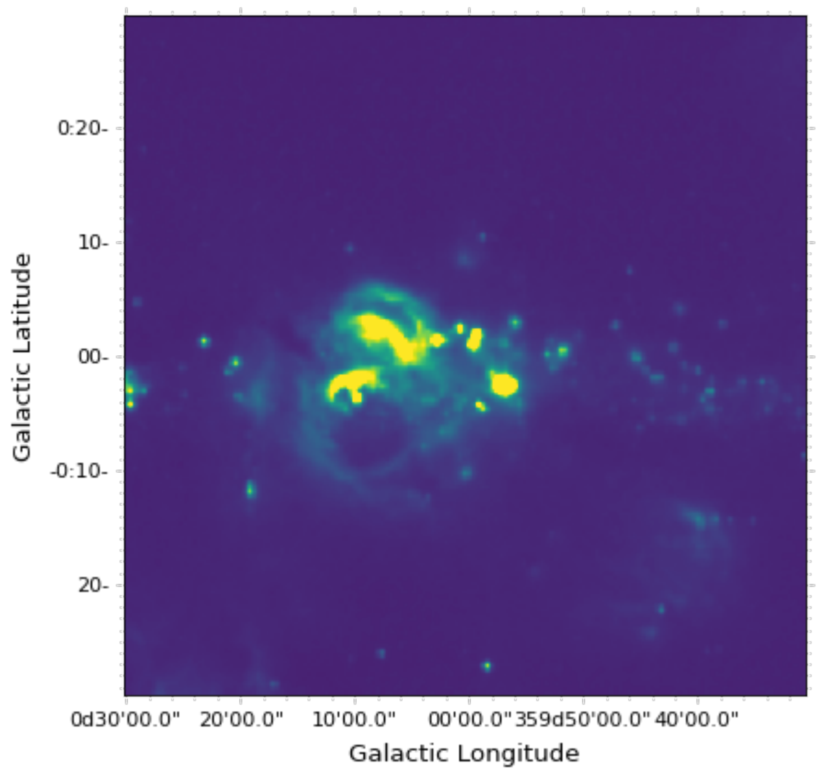
(png, svg, pdf)



And we can restore the ticks and tick labels again using:

```
lon.set_ticks_visible(True)
lon.set_ticklabel_visible(True)
lat.set_ticks_visible(True)
lat.set_ticklabel_visible(True)
lon.set_axislabel('Galactic Longitude')
lat.set_axislabel('Galactic Latitude')
```

(png, svg, pdf)

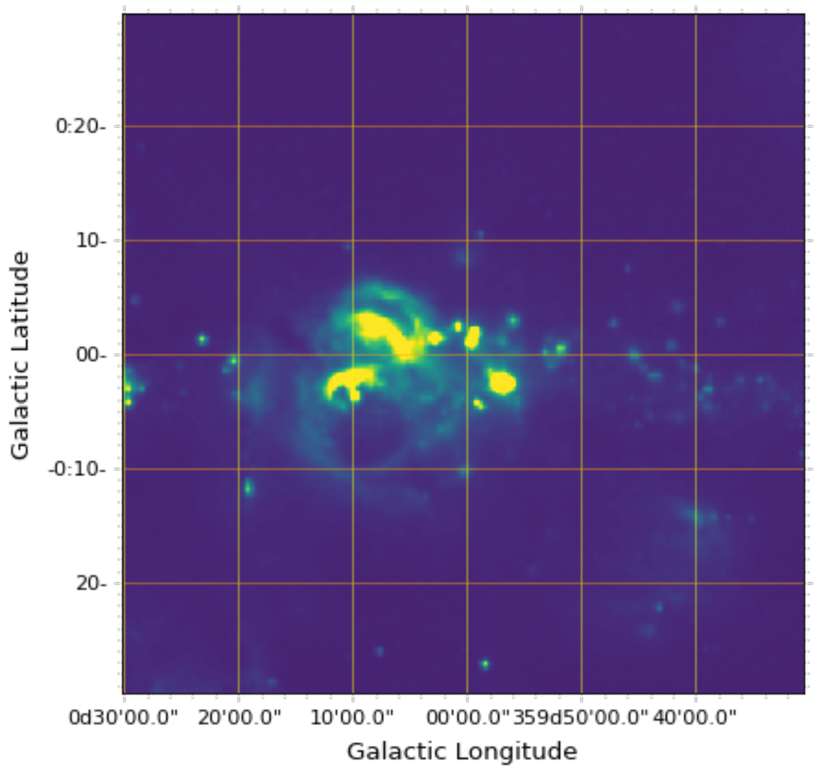


Coordinate grid

Since the properties of a coordinate grid are linked to the properties of the ticks and labels, grid lines 'belong' to the coordinate objects described above. For example, you can show a grid with yellow lines for RA and orange lines for declination with:

```
lon.grid(color='yellow', alpha=0.5, linestyle='solid')
lat.grid(color='orange', alpha=0.5, linestyle='solid')
```

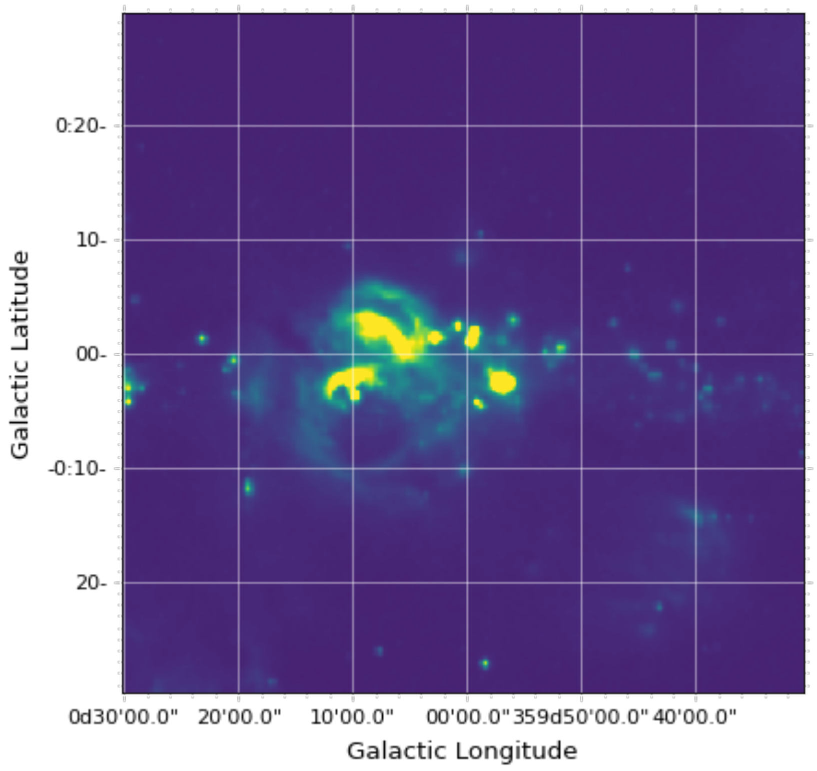
(png, svg, pdf)



For convenience, you can also simply draw a grid for all the coordinates in one command:

```
ax.coords.grid(color='white', alpha=0.5, linestyle='solid')
```

(png, svg, pdf)



Note

If you use the pyplot interface, you can also plot the grid using `plt.grid()`.

Overplotting markers and artists

For the example in the following page we start from the example introduced in Initializing axes with world coordinates.

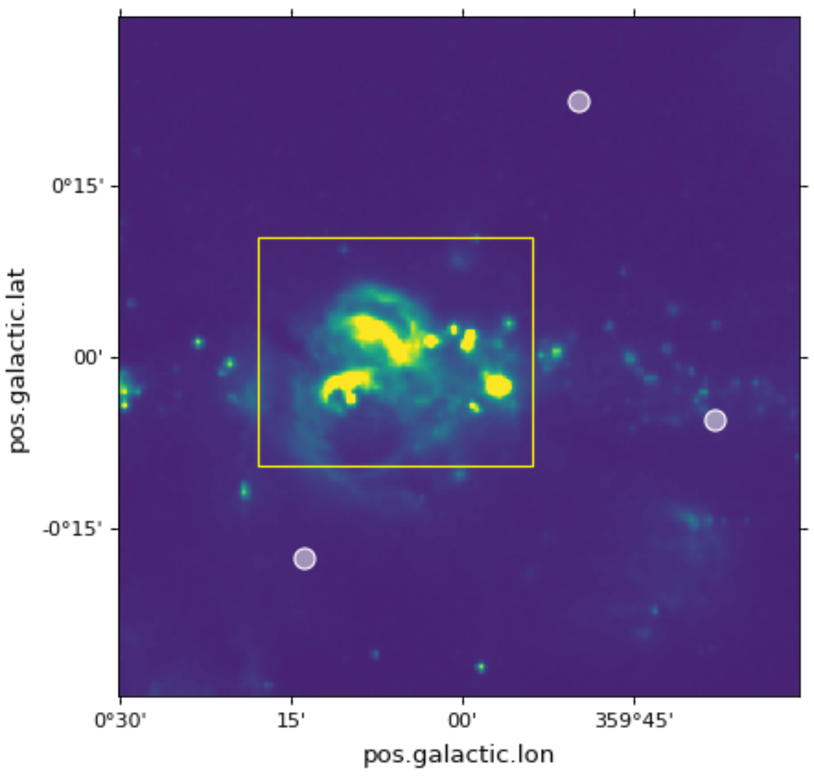
Pixel coordinates

Apart from the handling of the ticks, tick labels, and grid lines, the `WCSAxes` class behaves like a normal Matplotlib `Axes` instance, and methods such as `imshow()`, `contour()`, `plot()`, `scatter()`, and so on will work and plot the data in **pixel coordinates** by default.

In the following example, the scatter markers and the rectangle will be plotted in pixel coordinates:

```
# The following line makes it so that the zoom level no longer
changes,
# otherwise Matplotlib has a tendency to zoom out when adding
overlays.
ax.set_autoscale_on(False)

# Add a rectangle with bottom left corner at pixel position (30, 50)
with a
# width and height of 60 and 50 pixels respectively.
from matplotlib.patches import Rectangle
r = Rectangle((30., 50.), 60., 50., edgecolor='yellow',
facecolor='none')
ax.add_patch(r)

# Add three markers at (40, 30), (100, 130), and (130, 60). The
facecolor is
# a transparent white (0.5 is the alpha value).
ax.scatter([40, 100, 130], [30, 130, 60], s=100, edgecolor='white',
facecolor=(1, 1, 1, 0.5))
```

(png, svg, pdf)



World coordinates

All such Matplotlib commands allow a `transform=` argument to be passed, which will transform the input from world to pixel coordinates before it is passed to Matplotlib and plotted. For instance:

```
ax.scatter(..., transform=...)
```

will take the values passed to `scatter()` and will transform them using the transformation passed to `transform=`, in order to end up with the final pixel coordinates.

The `WCSAxes` class includes a `get_transform()` method that can be used to get the appropriate transformation object to convert from various world coordinate systems to the final pixel coordinate system required by Matplotlib. The `get_transform()` method can take a number of different inputs, which are described in this and subsequent sections. The two simplest inputs to this method are `'world'` and `'pixel'`.

For example, if your WCS defines an image where the coordinate system consists of an angle in degrees and a wavelength in nanometers, you can do:

```
ax.scatter([34], [3.2], transform=ax.get_transform('world'))
```

to plot a marker at (34deg, 3.2nm).

Using `ax.get_transform('pixel')` is equivalent to not using any transformation at all (and things then behave as described in the Pixel coordinates section).

Celestial coordinates

For the special case where the WCS represents celestial coordinates, a number of other inputs can be passed to `get_transform()`. These are:

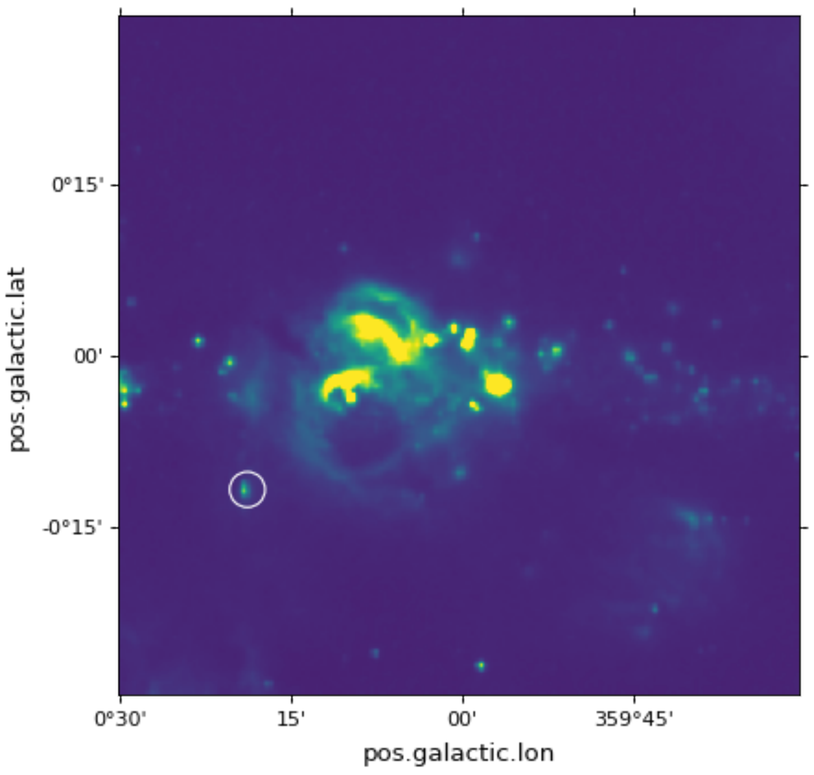
- `'fk4'` : B1950 FK4 equatorial coordinates
- `'fk5'` : J2000 FK5 equatorial coordinates
- `'icrs'` : ICRS equatorial coordinates
- `'galactic'` : Galactic coordinates

In addition, any valid `astropy.coordinates` coordinate frame can be passed.

For example, you can add markers with positions defined in the FK5 system using:

```
ax.scatter(266.78238, -28.769255, transform=ax.get_transform('fk5'),
s=300,
           edgecolor='white', facecolor='none')
```

(png, svg, pdf)



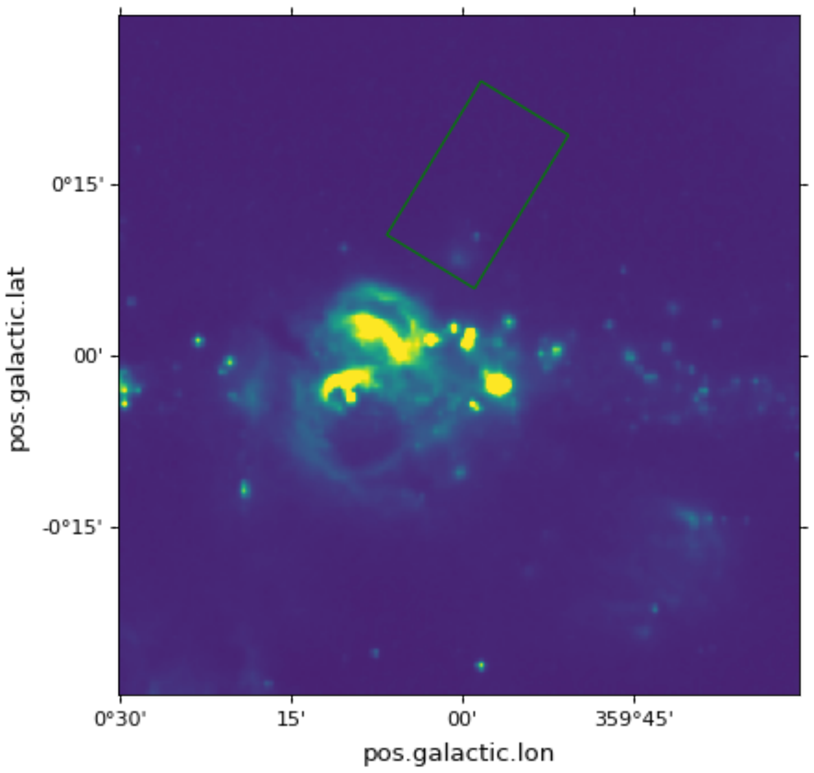
In the case of `scatter()` and `plot()`, the positions of the center of the markers is transformed, but the markers themselves are drawn in the frame of reference of the image, which means that they will not look distorted.

Patches/shapes/lines

Transformations can also be passed to Astropy or Matplotlib patches. For example, we can use the `get_transform()` method above to plot a quadrangle in FK5 equatorial coordinates:

```
from astropy import units as u
from astropy.visualization.wcsaxes import Quadrangle

r = Quadrangle((266.0, -28.9)*u.deg, 0.3*u.deg, 0.15*u.deg,
               edgecolor='green', facecolor='none',
               transform=ax.get_transform('fk5'))
ax.add_patch(r)
```

(png, svg, pdf)



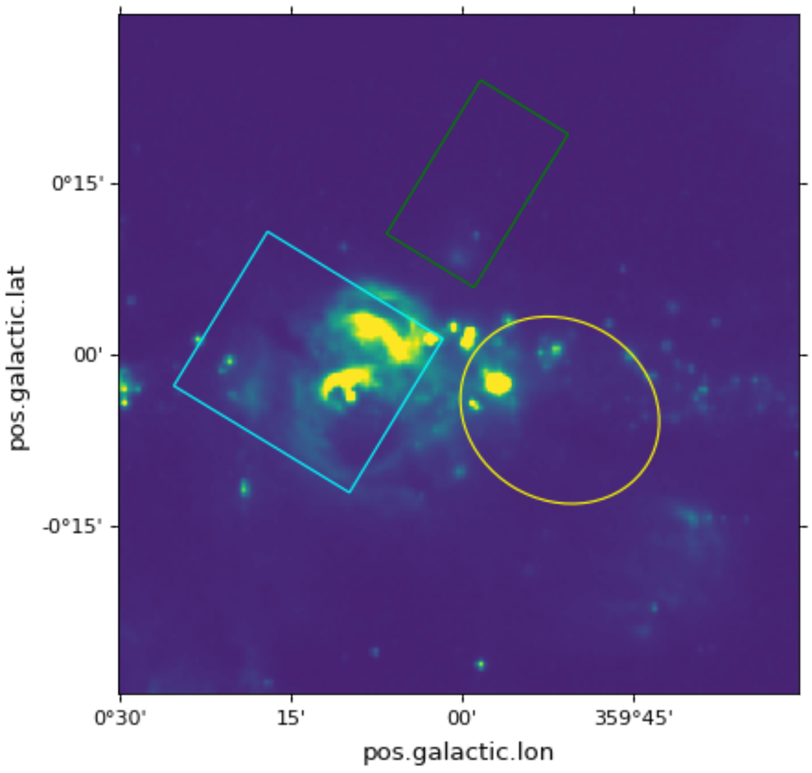
In this case, the quadrangle will be plotted at FK5 J2000 coordinates (266deg, -28.9deg). See the Quadrangles section for more information on `Quadrangle`.

However, it is **very important** to note that while the height will indeed be 0.15 degrees, the width will not strictly represent 0.3 degrees on the sky, but an interval of 0.3 degrees in longitude (which, depending on the latitude, will represent a different angle on the sky). In other words, if the width and height are set to the same value, the resulting polygon will not be a square. The same applies to the `Circle` patch, which will not actually produce a circle:

```
from matplotlib.patches import Circle
```

```
r = Quadrangle((266.4, -28.9)*u.deg, 0.3*u.deg, 0.3*u.deg,
              edgecolor='cyan', facecolor='none',
              transform=ax.get_transform('fk5'))
ax.add_patch(r)

c = Circle((266.4, -29.1), 0.15, edgecolor='yellow',
facecolor='none',
           transform=ax.get_transform('fk5'))
ax.add_patch(c)
```

(png, svg, pdf)



Important

If what you are interested is simply plotting circles around sources to highlight them, then we recommend using **scatter()**, since for the circular marker (the default), the circles will be guaranteed to be circles in the plot, and only the position of the center is transformed.

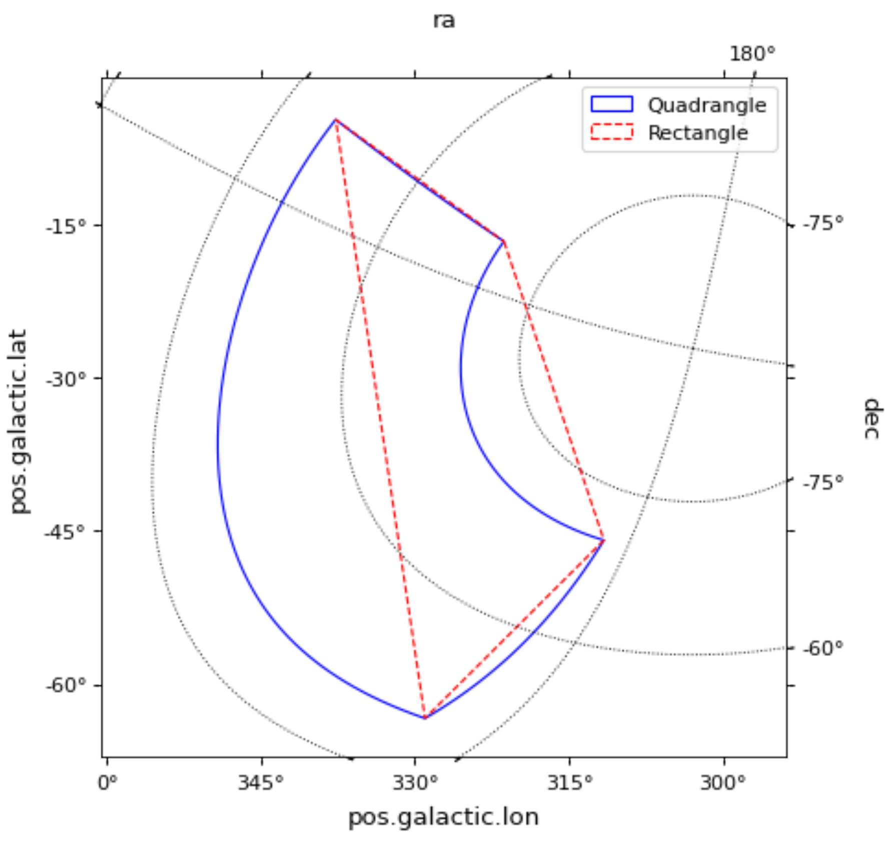
To plot 'true' spherical circles, see the Spherical patches section.

Quadrangles

Quadrangle is the recommended patch for plotting a quadrangle, as opposed to Matplotlib's **Rectangle**. The edges of a quadrangle lie on two lines of constant longitude and two lines of constant latitude (or the equivalent component names in the coordinate frame of interest, such as right ascension and declination). The edges of `Quadrangle` will render as curved lines if appropriate for the WCS transformation. In contrast, `Rectangle` will always have straight edges. Here's a comparison of the two types of patches for plotting a quadrangle in `ICRS` coordinates on `Galactic` axes:

```
from matplotlib.patches import Rectangle

# Set the Galactic axes such that the plot includes the ICRS south
pole
ax = plt.subplot(projection=wcs)
ax.set_xlim(0, 10000)
ax.set_ylim(-10000, 0)

# Overlay the ICRS coordinate grid
overlay = ax.get_coords_overlay('icrs')
overlay.grid(color='black', ls='dotted')

# Add a quadrangle patch (100 degrees by 20 degrees)
q = Quadrangle((255, -70)*u.deg, 100*u.deg, 20*u.deg,
               label='Quadrangle', edgecolor='blue',
facecolor='none',
               transform=ax.get_transform('icrs'))
ax.add_patch(q)

# Add a rectangle patch (100 degrees by 20 degrees)
r = Rectangle((255, -70), 100, 20,
              label='Rectangle', edgecolor='red', facecolor='none',
linestyle='--',
              transform=ax.get_transform('icrs'))
ax.add_patch(r)

plt.legend(loc='upper right')
```

(png, svg, pdf)

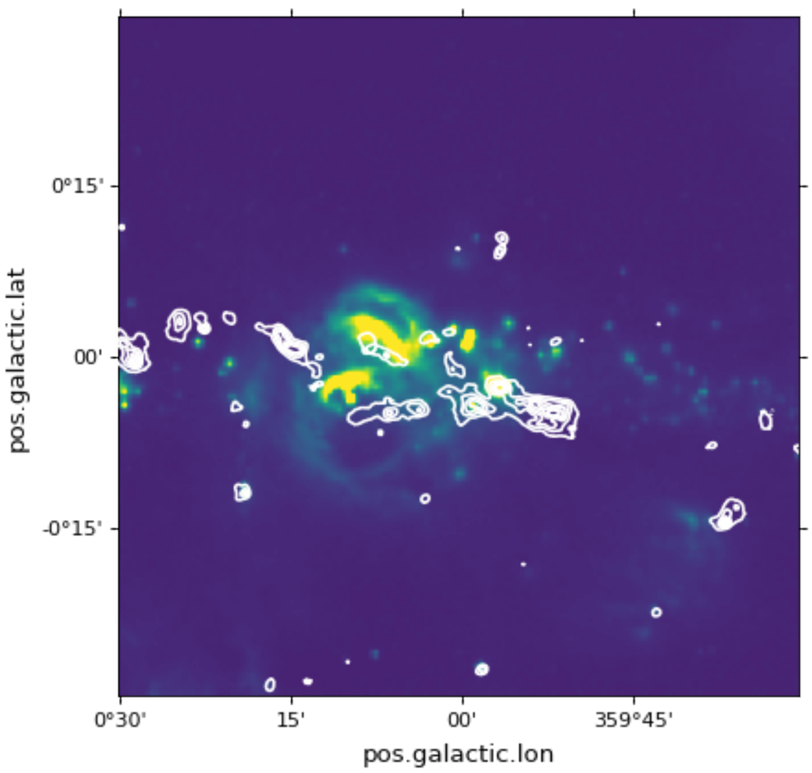


Contours

Overplotting contours is also simple using the **`get_transform()`** method. For contours, **`get_transform()`** should be given the WCS of the image to plot the contours for:

```
filename =
get_pkg_data_filename('galactic_center/gc_bolocam_gps.fits')
hdu = fits.open(filename)[0]
ax.contour(hdu.data, transform=ax.get_transform(WCS(hdu.header)),
           levels=[1,2,3,4,5,6], colors='white')
```

(png, svg, pdf)

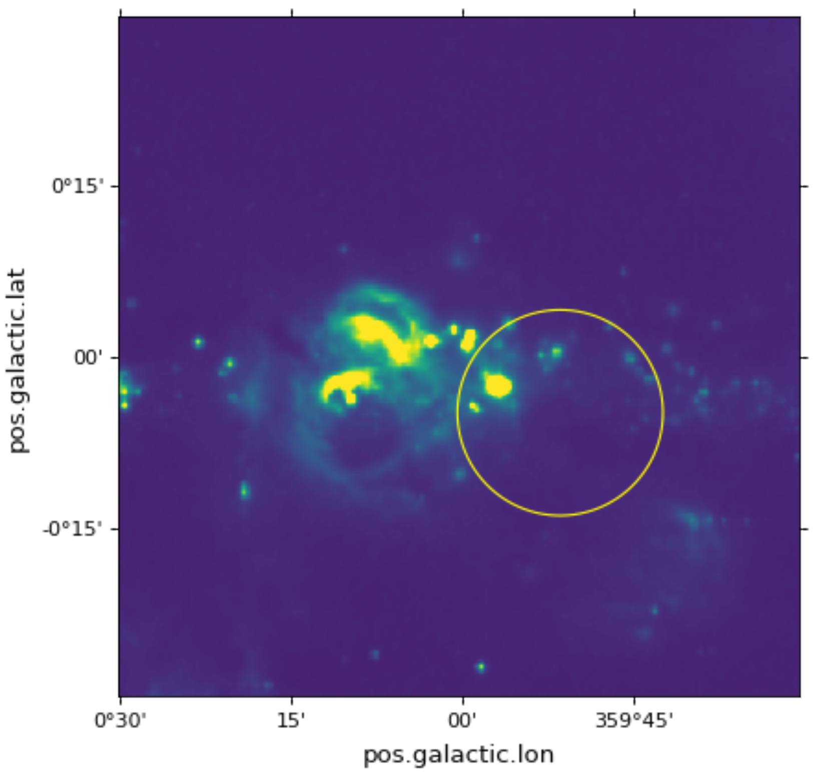


Spherical patches

In the case where you are making a plot of a celestial image, and want to plot a circle that represents the area within a certain angle of a longitude/latitude, the **Circle** patch is not appropriate, since it will result in a distorted shape (because longitude is not the same as the angle on the sky). For this use case, you can instead use **SphericalCircle**, which takes a tuple of **Quantity** as the input, and a **Quantity** as the radius:

```
from astropy import units as u
from astropy.visualization.wcsaxes import SphericalCircle

r = SphericalCircle((266.4 * u.deg, -29.1 * u.deg), 0.15 * u.degree,
                     edgecolor='yellow', facecolor='none',
                     transform=ax.get_transform('fk5'))
ax.add_patch(r)
```

(png, svg, pdf)



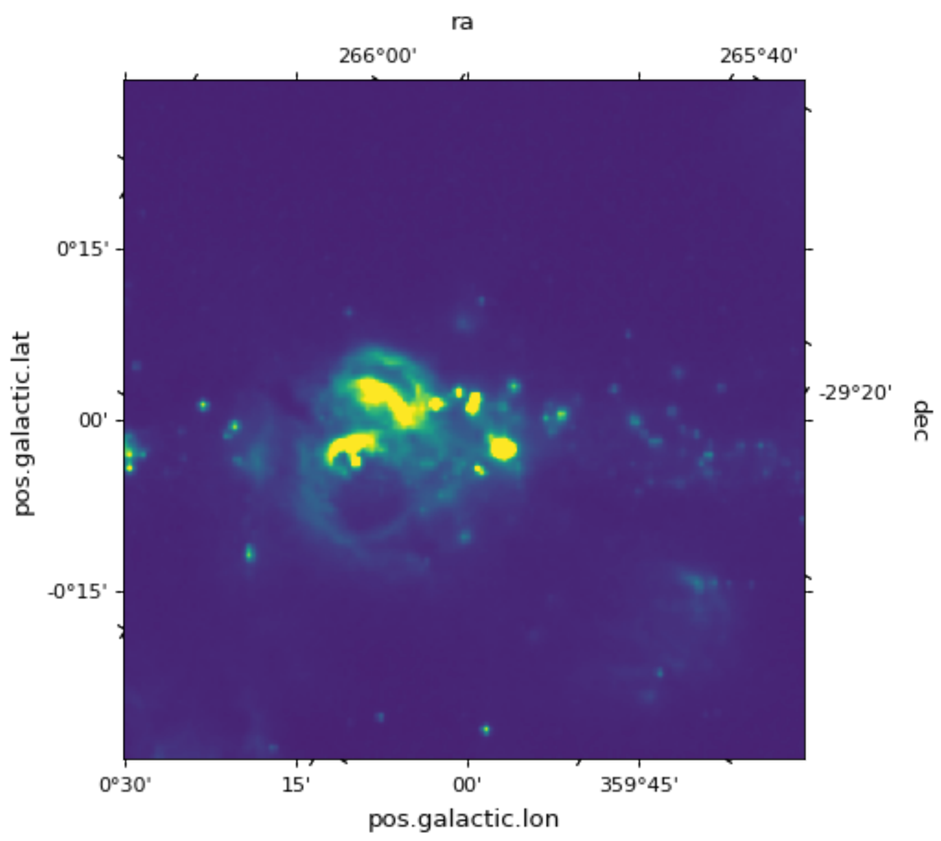
Overlaying coordinate systems

For the example in the following page we start from the example introduced in Initializing axes with world coordinates.

The coordinates shown by default in a plot will be those derived from the WCS or transformation passed to the **`WCSAxes`** class. However, it is possible to overlay different coordinate systems using the **`get_coords_overlay()`** method:

```
overlay = ax.get_coords_overlay('fk5')
```

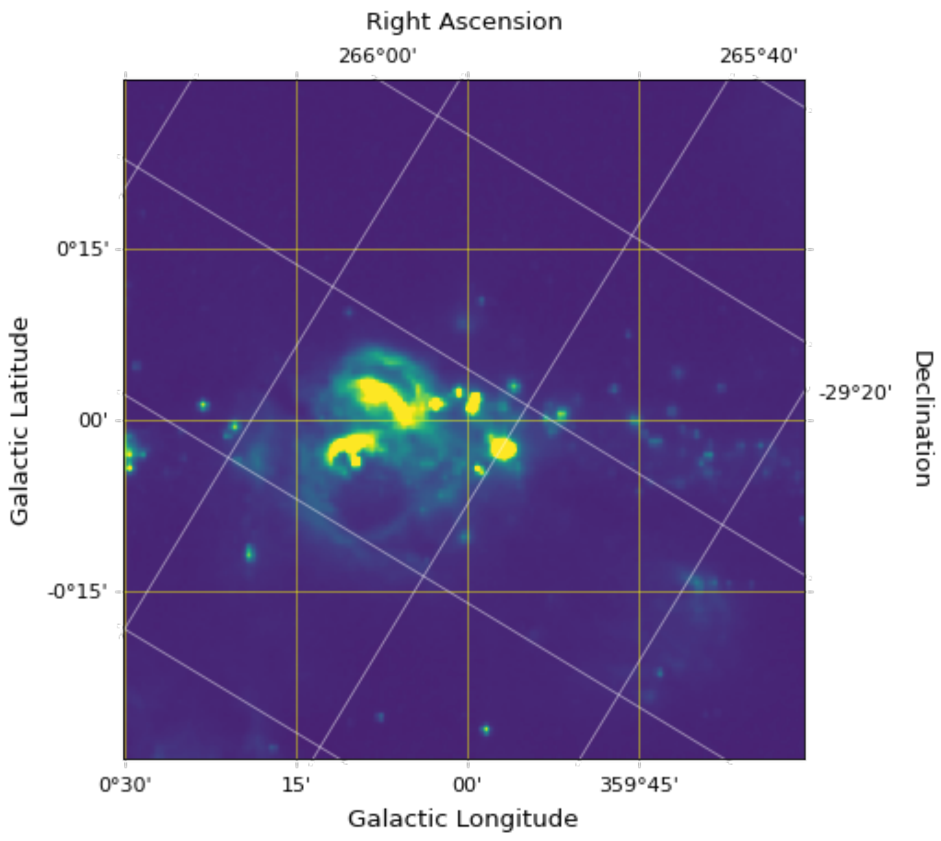
(png, svg, pdf)



The object returned is a **`CoordinatesMap`**, the same type of object as `ax.coord`. It can therefore be used in the same way as `ax.coord` to set the ticks, tick labels, and axis labels properties:

```
ax.coords['glon'].set_ticks(color='white')
ax.coords['glat'].set_ticks(color='white')

ax.coords['glon'].set_axislabel('Galactic Longitude')
ax.coords['glat'].set_axislabel('Galactic Latitude')

ax.coords.grid(color='yellow', linestyle='solid', alpha=0.5)

overlay['ra'].set_ticks(color='white')
overlay['dec'].set_ticks(color='white')

overlay['ra'].set_axislabel('Right Ascension')
overlay['dec'].set_axislabel('Declination')

overlay.grid(color='white', linestyle='solid', alpha=0.5)
```

(png, svg, pdf)



Slicing Multidimensional Data
WCSAxes can either plot one or two dimensional data. If we have a dataset with higher dimensionality than the plot we want to make, we have to select which dimensions to use for the x or x and y axes of the plot. This example will show how to slice a FITS data cube and plot an image from it.

Slicing the WCS object

Like the example introduced in Initializing axes with world coordinates, we will read in the data using astropy.io.fits and parse the WCS information. The original FITS file can be downloaded from here.

```
import matplotlib.pyplot as plt
from astropy.wcs import WCS
from astropy.io import fits
from astropy.utils.data import get_pkg_data_filename
filename = get_pkg_data_filename('l1448/l1448_13co.fits')
hdu = fits.open(filename)[0]
wcs = WCS(hdu.header)
image_data = hdu.data
```

This is a three-dimensional dataset which you can check by looking at the header information by:

```
>>> hdu.header
```

```
...
NAXIS = 3 /number of axes
CTYPE1  = 'RA---SFL'          /
CTYPE2  = 'DEC--SFL'          /
CTYPE3  = 'VELO-LSR'          /
...
```

The header keyword ‘NAXIS’ gives the number of dimensions of the dataset. The keywords ‘CTYPE1’, ‘CTYPE2’ and ‘CTYPE3’ give the data type of these dimensions to be right ascension, declination and velocity respectively.

We then instantiate the **WCSAxes** using the **WCS** object and select the slices we want to plot:

```
import matplotlib.pyplot as plt
ax = plt.subplot(projection=wcs, slices=(50, 'y', 'x'))
```

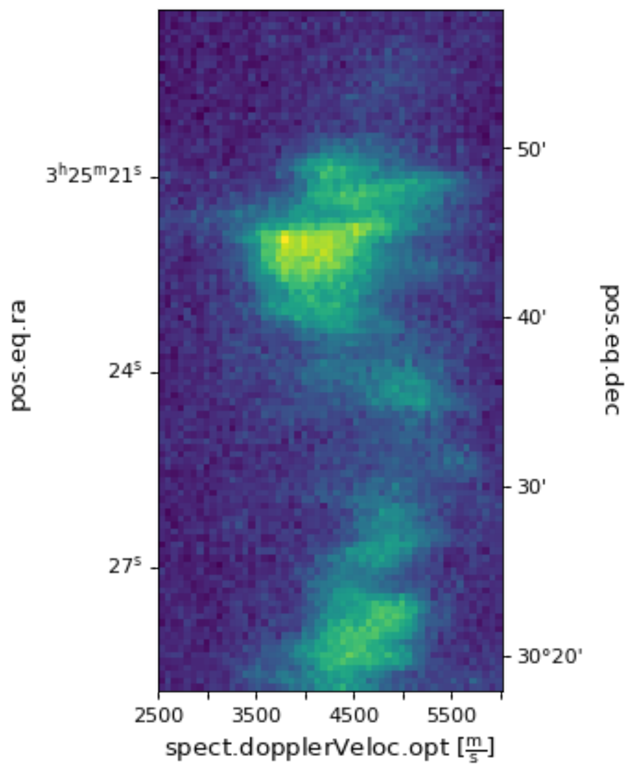
By setting `slices=(50, 'y', 'x')`, we have chosen to plot the second dimension on the y-axis and the third dimension on the x-axis. Even though we are not plotting the all the dimensions, we have to specify which slices to select for the dimensions that are not shown. In this example, we are not plotting the first dimension so we have selected the slice 50 to display. You can experiment with this by changing the selected slice and looking at how the plotted image changes.

Plotting the image

We then add the axes to the image and plot it using the method **imshow()**.

```
ax.coords[2].set_ticklabel(exclude_overlapping=True)
ax.imshow(image_data[:, :, 50].transpose())
```

(png, svg, pdf)

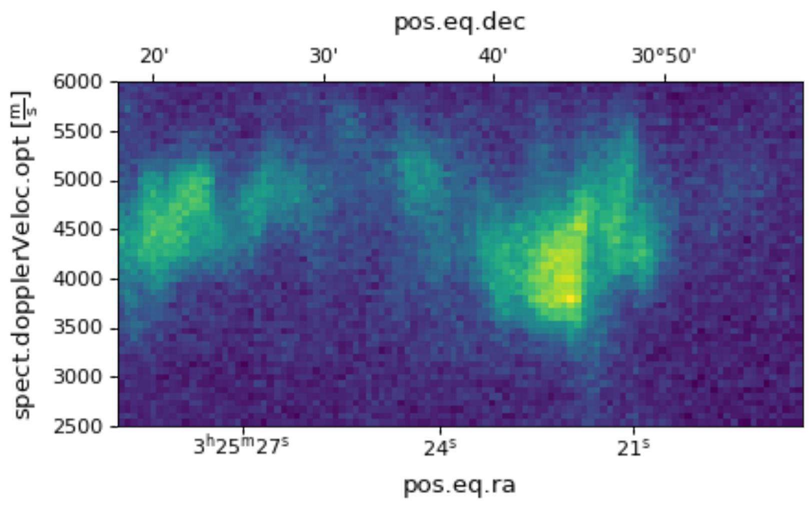


Here, `image_data` is an **ndarray** object. In Numpy, the order of the axes is reversed so the first dimension in the FITS file appears last, the last dimension appears first and so on. Therefore the index passed to **imshow()** should be the same as passed to `slices` but in reversed order. We also need to **transpose()** `image_data` as we have reversed the dimensions plotted on the x and y axes in the slice.

If we don't want to reverse the dimensions plotted, we can simply do:

```
import matplotlib.pyplot as plt
ax = plt.subplot(projection=wcs, slices=(50, 'x', 'y'))
ax.imshow(image_data[:, :, 50])
```

(png, svg, pdf)



Plotting one dimensional data

If we wanted to plot the spectral axes for one pixel we can do this by slicing down to one dimension.

```
import matplotlib.pyplot as plt
ax = plt.subplot(projection=wcs, slices=(50, 50, 'x'))
```

Here we have selected the 50 pixel in the first and second dimensions and will use the third dimension as our x axis.

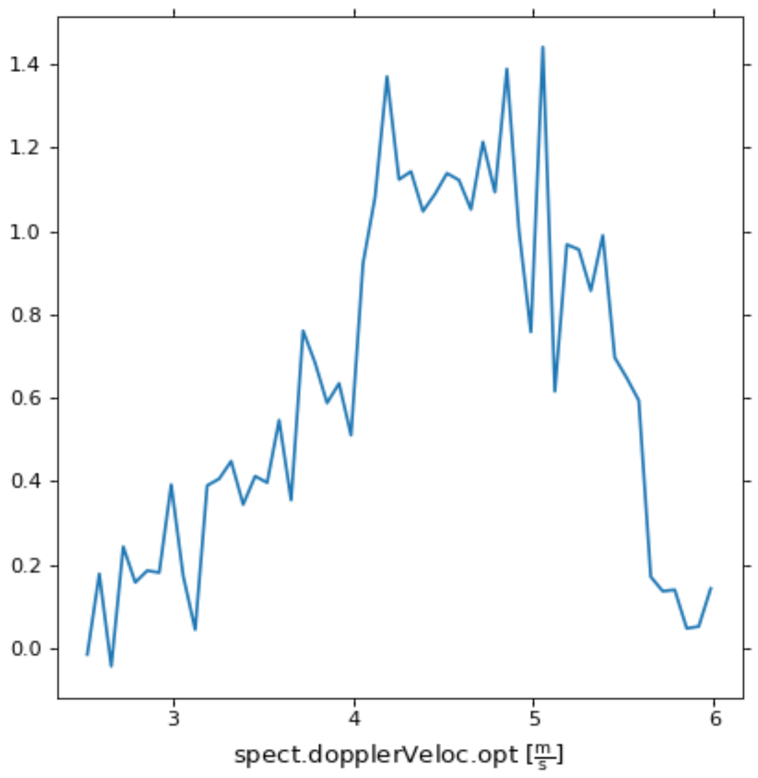
We can now plot the spectral axis for this pixel. Note that we are plotting against pixel coordinates in the call to `ax.plot`, `WCSAxes` will display the world coordinates for us.

```
ax.plot(image_data[:, 50, 50])
```

As this is still a `WCSAxes` plot, we can set the display units for the x-axis

```
ra, dec, vel = ax.coords
vel.set_format_unit(u.km/u.s)
```

(png, svg, pdf)

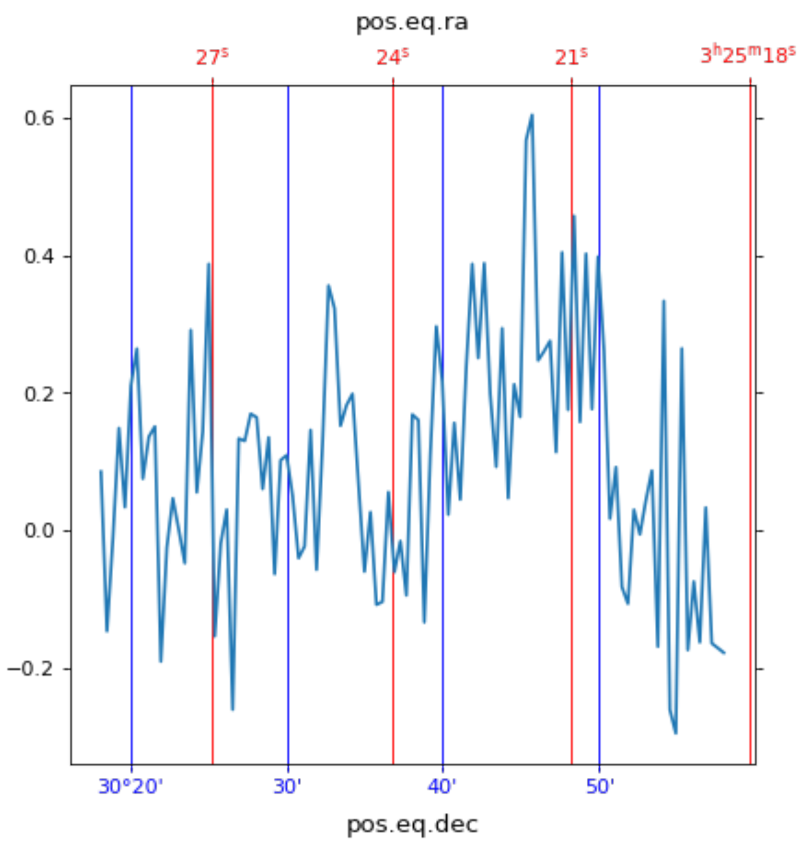


If we wanted to plot a one dimensional plot along a spatial dimension, i.e. intensity along a row in the image, `WCSAxes` defaults to displaying both the world coordinates for this plot. We can customise the colors and add grid lines for each of the spatial axes.

```
import matplotlib.pyplot as plt
ax = plt.subplot(projection=wcs, slices=(50, 'x', 0))
```

```
ax.plot(image_data[0, :, 50])

ra, dec, wave = ax.coords
ra.set_ticks(color="red")
ra.set_ticklabel(color="red")
ra.grid(color="red")

dec.set_ticks(color="blue")
dec.set_ticklabel(color="blue")
dec.grid(color="blue")
```

(png, svg, pdf)



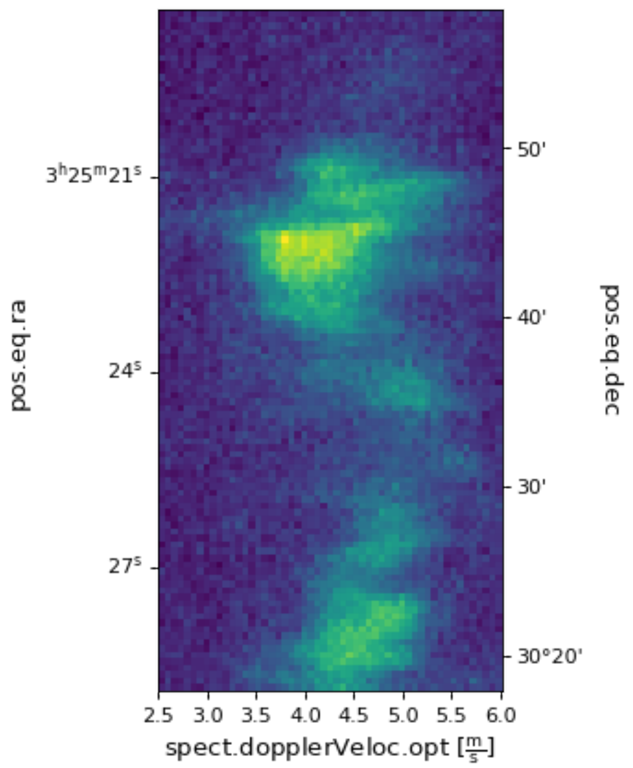
Controlling Axes

Changing Axis Units

WCSAxes also allows users to change the units of the axes of an image. In the example in Slicing Multidimensional Data, the x axis represents velocity in m/s. We can change the unit to an equivalent one by:

```
import astropy.units as u
ax.coords[2].set_major_formatter('x.x') # Otherwise values round to the nearest whole number
ax.coords[2].set_format_unit(u.km / u.s)
```

(png, svg, pdf)



Disabling Automatic Labelling

By default WCSAxes adds labels to the axes to indicate what world coordinate is being represented on that axis, and what unit is being used to display it. If you want to disable this behavior you can either set an explicit label for that axis with `set_axislabel` or you can disable the feature per coordinate with:

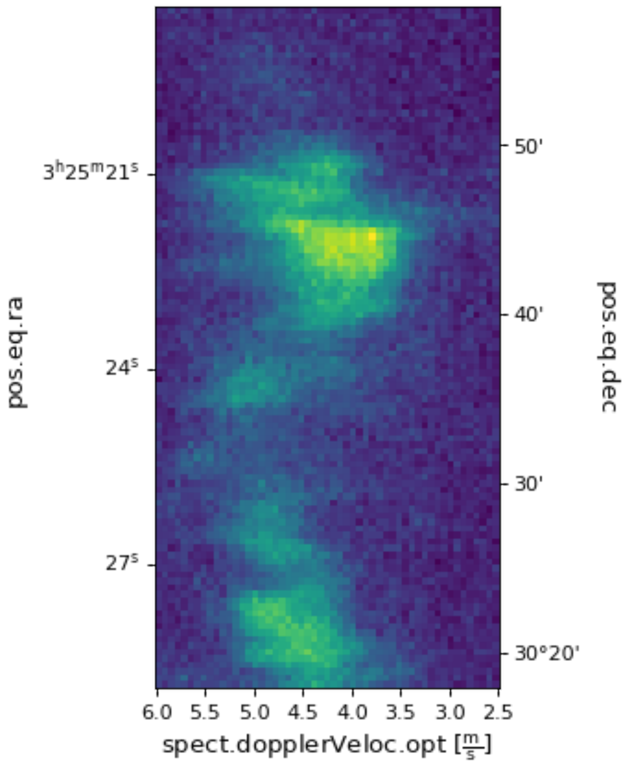
```
ax = plt.subplot(projection=wcs)  # doctest: +SKIP
ax.coords[0].set_auto_axislabel(False)  # doctest: +SKIP
```

Changing Axis Directions

Sometimes astronomy FITS files don't follow the convention of having the longitude increase to the left, so we want to flip an axis so that it goes in the opposite direction. To do this on our example image:

```
ax.invert_xaxis()
```

(png, svg, pdf)



Initializing WCSAxes with custom transforms

In Initializing axes with world coordinates, we saw how to make plots using `WCS` objects. However, the `WCSAxes` class can also be initialized with more general transformations that don't have to be represented by the `WCS` class. Instead, you can initialize `WCSAxes` using a Matplotlib `Transform` object and a dictionary (`coord_meta`) that provides metadata on how to interpret the transformation.

The `Transform` should represent the conversion from pixel to world coordinates, and should have `input_dims=2` and can have `output_dims` set to any positive integer. In addition, `has_inverse` should be set to `True` and the `inverted` method should be implemented.

The `coord_meta` dictionary should include the following keys:

- `name`: an iterable of strings giving the names for each dimension
- `type`: an iterable of strings that should be either `'longitude'`, `'latitude'`, or `'scalar'` (for anything that isn't a longitude or latitude).
- `wrap`: an iterable of values which indicate for longitudes at which angle (in degrees) to wrap the coordinates. This should be `None` unless `type` is `'longitude'`.
- `unit`: an iterable of `Unit` objects giving the units of the world coordinates returned by the `Transform`.
- `format_unit`: an iterable of `Unit` objects giving the units to use for the

formatting of the labels. These can be set to **None** to default to the units given in `unit`, but can be set for example if the **Transform** returns values in degrees and you want the labels to be formatted in hours.

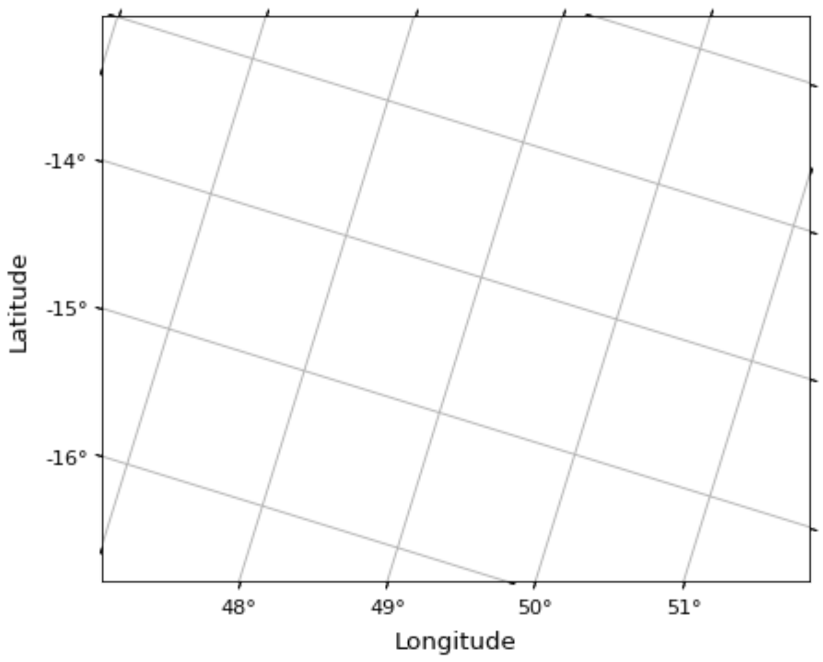
In addition the `coord_meta` can optionally include the following keys:

- `default_axislabel_position`: an iterable of strings giving for each world coordinates the spine of the frame on which to show the axis label for the coordinate. Each string should be such that it could be used as input to **set_axislabel_position()**.
- `default_ticklabel_position`: an iterable of strings giving for each world coordinates the spine of the frame on which to show the tick labels for the coordinate. Each string should be such that it could be used as input to **set_ticklabel_position()**.
- `default_ticks_position`: an iterable of strings giving for each world coordinates the spine of the frame on which to show the ticks for the coordinate. Each string should be such that it could be used as input to **set_ticks_position()**.

The following example illustrates a custom projection using a transform and `coord_meta`:

```
from astropy import units as u
import matplotlib.pyplot as plt
from matplotlib.transforms import Affine2D
from astropy.visualization.wcsaxes import WCSAxes

# Set up an affine transformation
transform = Affine2D()
transform.scale(0.01)
transform.translate(40, -30)
transform.rotate(0.3)  # radians

# Set up metadata dictionary
coord_meta = {}
coord_meta['name'] = 'lon', 'lat'
coord_meta['type'] = 'longitude', 'latitude'
coord_meta['wrap'] = 180, None
coord_meta['unit'] = u.deg, u.deg
coord_meta['format_unit'] = None, None

fig = plt.figure()
ax = WCSAxes(fig, [0.1, 0.1, 0.8, 0.8], aspect='equal',
             transform=transform, coord_meta=coord_meta)
fig.add_axes(ax)
ax.set_xlim(-0.5, 499.5)
ax.set_ylim(-0.5, 399.5)
ax.grid()
ax.coords['lon'].set_axislabel('Longitude')
ax.coords['lat'].set_axislabel('Latitude')
```

(png, svg, pdf)



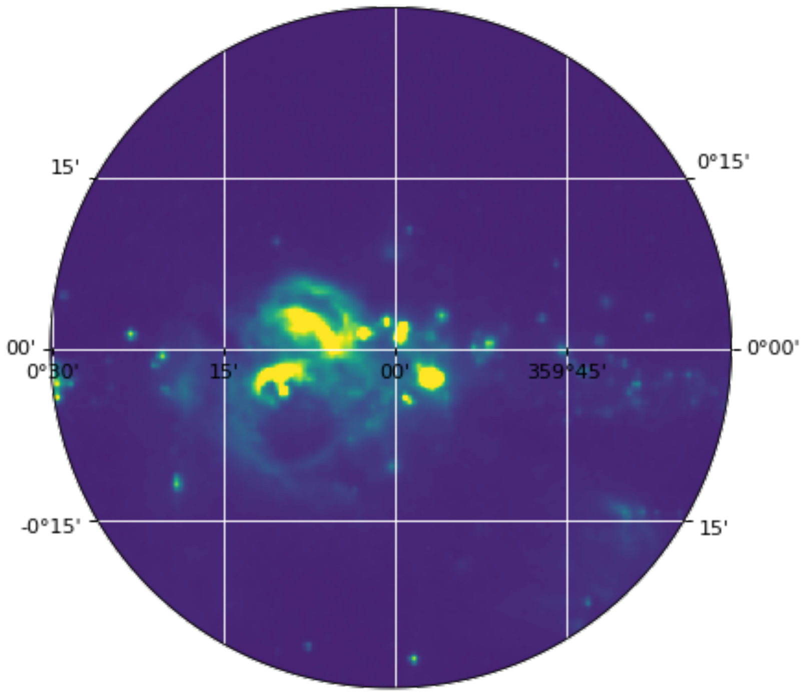
Using a custom frame

By default, **WCSAxes** will make use of a rectangular frame for a plot, but this can be changed to provide any custom frame. The following example shows how to use the built-in **EllipticalFrame** class, which is an ellipse which extends to the same limits as the built-in rectangular frame:

```
from astropy.wcs import WCS
from astropy.io import fits
from astropy.utils.data import get_pkg_data_filename
from astropy.visualization.wcsaxes.frame import EllipticalFrame
import matplotlib.pyplot as plt

filename = get_pkg_data_filename('galactic_center/gc_msx_e.fits')
hdu = fits.open(filename)[0]
wcs = WCS(hdu.header)

ax = plt.subplot(projection=wcs, frame_class=EllipticalFrame)

ax.coords.grid(color='white')

im = ax.imshow(hdu.data, vmin=-2.e-5, vmax=2.e-4, origin='lower')
```

```
# Clip the image to the frame
im.set_clip_path(ax.coords.frame.patch)
```

(png, svg, pdf)

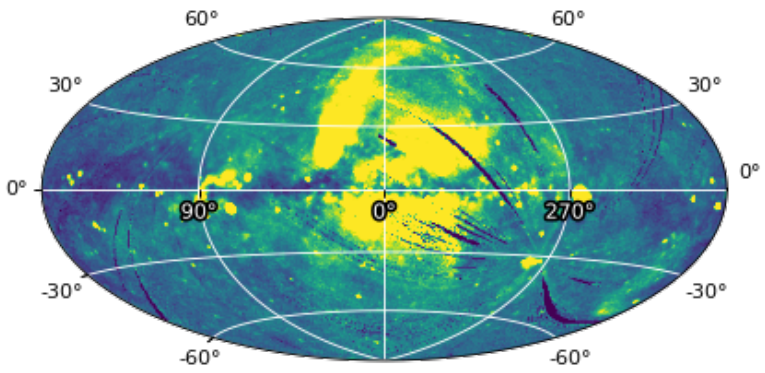


The `EllipticalFrame` class is especially useful for all-sky plots such as Aitoff projections:

```
from astropy.wcs import WCS
from astropy.io import fits
from astropy.utils.data import get_pkg_data_filename
from astropy.visualization.wcsaxes.frame import EllipticalFrame
from matplotlib import patheffects
import matplotlib.pyplot as plt

filename = get_pkg_data_filename('allsky/allsky_rosat.fits')
hdu = fits.open(filename)[0]
wcs = WCS(hdu.header)

ax = plt.subplot(projection=wcs, frame_class=EllipticalFrame)

path_effects=[patheffects.withStroke(linewidth=3,
foreground='black')]
ax.coords.grid(color='white')
ax.coords['glon'].set_ticklabel(color='white',
path_effects=path_effects)

im = ax.imshow(hdu.data, vmin=0., vmax=300., origin='lower')
```

```
# Clip the image to the frame
im.set_clip_path(ax.coords.frame.patch)
```

(png, svg, pdf)



However, you can also write your own frame class. The idea is to set up any number of connecting spines that define the frame. You can define a frame as a spine, but if you define it as multiple spines you will be able to control on which spine the tick labels and ticks should appear.

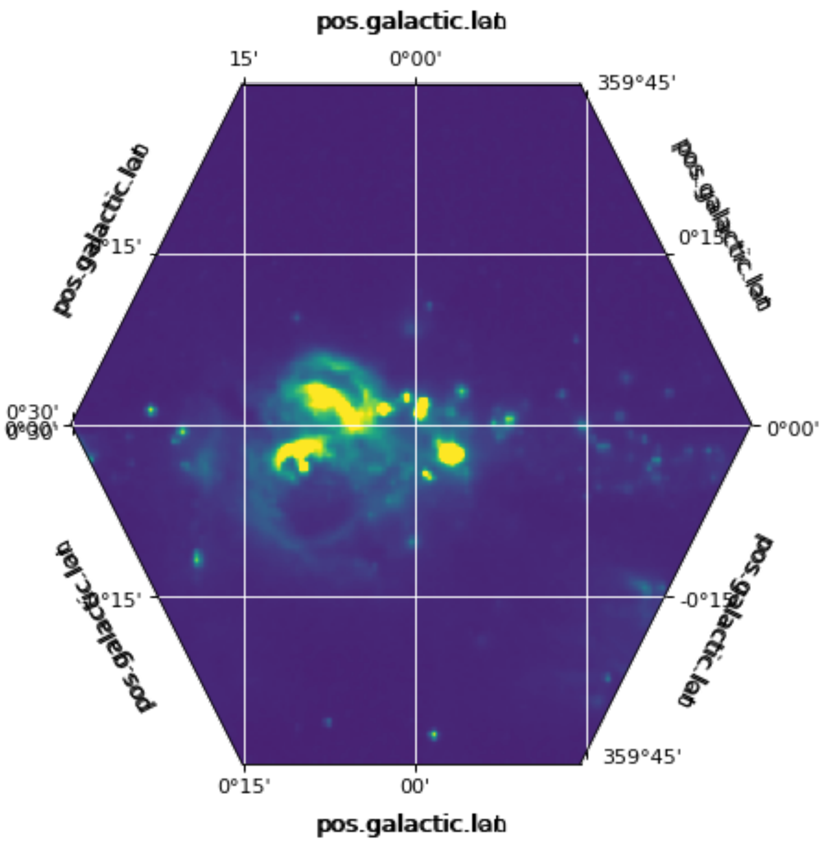
The following example shows how you could for example define a hexagonal frame:

```
import numpy as np
from astropy.visualization.wcsaxes.frame import BaseFrame

class HexagonalFrame(BaseFrame):

    spine_names = 'abcdef'

    def update_spines(self):

        xmin, xmax = self.parent_axes.get_xlim()
        ymin, ymax = self.parent_axes.get_ylim()

        ymid = 0.5 * (ymin + ymax)
        xmid1 = (xmin + xmax) / 4.
        xmid2 = (xmin + xmax) * 3. / 4.

        self['a'].data = np.array(([xmid1, ymin], [xmid2, ymin]))
        self['b'].data = np.array(([xmid2, ymin], [xmax, ymid]))
        self['c'].data = np.array(([xmax, ymid], [xmid2, ymax]))
        self['d'].data = np.array(([xmid2, ymax], [xmid1, ymax]))
        self['e'].data = np.array(([xmid1, ymax], [xmin, ymid]))
        self['f'].data = np.array(([xmin, ymid], [xmid1, ymin]))
```

which we can then use:

```
from astropy.wcs import WCS
from astropy.io import fits
from astropy.utils.data import get_pkg_data_filename
import matplotlib.pyplot as plt

filename = get_pkg_data_filename('galactic_center/gc_msx_e.fits')
hdu = fits.open(filename)[0]
wcs = WCS(hdu.header)

ax = plt.subplot(projection=wcs, frame_class=HexagonalFrame)

ax.coords.grid(color='white')

im = ax.imshow(hdu.data, vmin=-2.e-5, vmax=2.e-4, origin='lower')

# Clip the image to the frame
im.set_clip_path(ax.coords.frame.patch)
```

(png, svg, pdf)

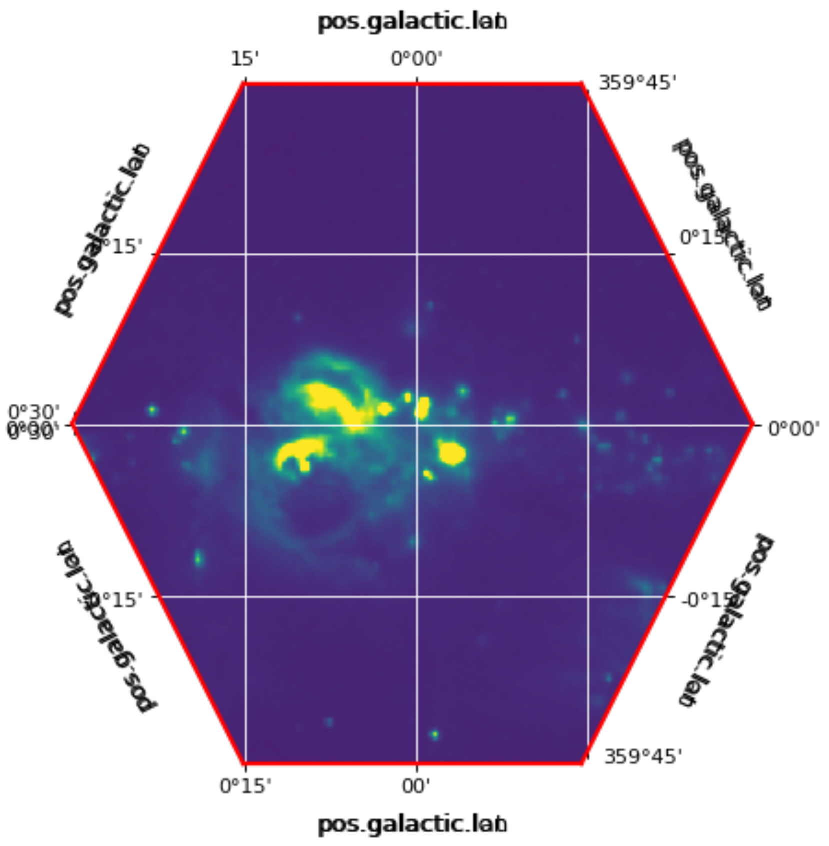


Frame properties

The color and linewidth of the frame can also be set by

```
ax.coords.frame.set_color('red')
ax.coords.frame.set_linewidth(2)
```

(png, svg, pdf)



Reference/API

astropy.visualization.wcsaxes Package

Classes

Conf()	Configuration parameters for **astropy.visualization.wcsaxes**.
CoordinateHelper([parent_axes, parent_map, …])	Helper class to control one of the coordinates in the **WCSAxes**.
CoordinatesMap(axes[, transform, …])	A container for coordinate helpers that represents a coordinate system.
Quadrangle(anchor, width, height[, …])	Create a patch representing a latitude-longitude quadrangle.
SphericalCircle(center, radius[, …])	Create a patch representing a spherical circle - that is, a circle that is formed of all the points that are within a certain angle of the central coordinates on a sphere.
WCSAxes(fig, rect[, wcs, transform, …])	The main axes class that can be used to show world coordinates from a WCS.
WCSAxesSubplot(fig, *args, **kwargs)	A subclass class for WCSAxes

astropy.visualization.wcsaxes.frame Module

Classes

`RectangularFrame1D`(parent_axes, transform[, …])	A classic rectangular frame.
`Spine`(parent_axes, transform)	A single side of an axes.
`BaseFrame`(parent_axes, transform[, path])	Base class for frames, which are collections of `Spine` instances.
`RectangularFrame`(parent_axes, transform[, path])	A classic rectangular frame.
`EllipticalFrame`(parent_axes, transform[, path])	An elliptical frame.

Image stretching and normalization

The `astropy.visualization` module provides a framework for transforming values in images (and more generally any arrays), typically for the purpose of visualization. Two main types of transformations are provided:

- Normalization to the [0:1] range using lower and upper limits where \(x\) represents the values in the original image:

\[y = \frac{x - v_{\rm min}}{v_{\rm max} - v_{\rm min}}\]

- *Stretching* of values in the [0:1] range to the [0:1] range using a linear or non-linear function:

\[z = f(y)\]

In addition, classes are provided in order to identify lower and upper limits for a dataset based on specific algorithms (such as using percentiles).

Identifying lower and upper limits, as well as re-normalizing, is described in the Intervals and Normalization section, while stretching is described in the Stretching section.

Intervals and Normalization

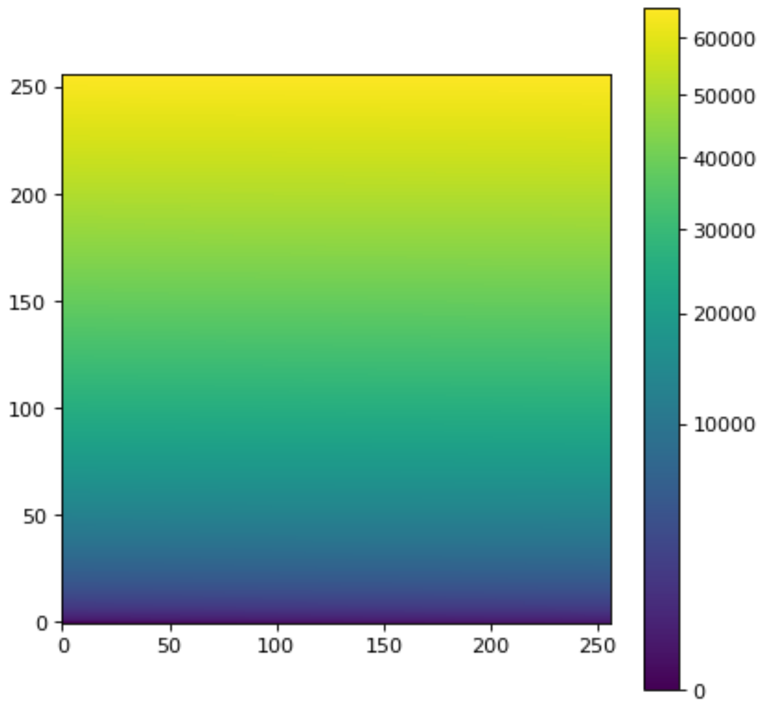
The Quick Way

`astropy` provides a convenience `simple_norm()` function that can be useful for quick interactive analysis:

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.visualization import simple_norm

# Generate a test image
image = np.arange(65536).reshape((256, 256))

# Create an ImageNormalize object
```

```
norm = simple_norm(image, 'sqrt')

# Display the image
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
im = ax.imshow(image, origin='lower', norm=norm)
fig.colorbar(im)
```

(png, svg, pdf)



This convenience function combines a **`Stretch`** object with an **`Interval`** object. We recommend using **`ImageNormalize`** directly in scripted programs instead of this convenience function.

The detailed way

Several classes are provided for determining intervals and for normalizing values in this interval to the [0:1] range. One of the simplest examples is the **`MinMaxInterval`** which determines the limits of the values based on the minimum and maximum values in the array. The class is instantiated with no arguments:

```
>>> from astropy.visualization import MinMaxInterval
>>> interval = MinMaxInterval()
```

and the limits can be determined by calling the **`get_limits()`** method, which takes the array of values:

```
>>> interval.get_limits([1, 3, 4, 5, 6])
(1, 6)
```

The `interval` instance can also be called like a function to actually normalize values to the range:

```
>>> interval([1, 3, 4, 5, 6])
array([0. , 0.4, 0.6, 0.8, 1. ])
```

Other interval classes include **`ManualInterval`**, **`PercentileInterval`**, **`AsymmetricPercentileInterval`**, and **`ZScaleInterval`**. For these, values in the array can fall outside of the limits given by the interval. A `clip` argument is provided to control the behavior of the normalization when values fall outside the limits:

```
>>> from astropy.visualization import PercentileInterval
>>> interval = PercentileInterval(50.)
>>> interval.get_limits([1, 3, 4, 5, 6])
(3.0, 5.0)
>>> interval([1, 3, 4, 5, 6])  # default is clip=True
array([0. , 0. , 0.5, 1. , 1. ])
>>> interval([1, 3, 4, 5, 6], clip=False)
array([-1. ,  0. ,  0.5,  1. ,  1.5])
```

Stretching

In addition to classes that can scale values to the [0:1] range, a number of classes are provided to ‘stretch’ the values using different functions. These map a [0:1] range onto a transformed [0:1] range. A simple example is the **`SqrtStretch`** class:

```
>>> from astropy.visualization import SqrtStretch
>>> stretch = SqrtStretch()
>>> stretch([0., 0.25, 0.5, 0.75, 1.])
array([0.        , 0.5       , 0.70710678, 0.8660254 , 1.        ])
```

As for the intervals, values outside the [0:1] range can be treated differently depending on the `clip` argument. By default, output values are clipped to the [0:1] range:

```
>>> stretch([-1., 0., 0.5, 1., 1.5])
array([0.        , 0.        , 0.70710678, 1.        , 1.        ])
```

but this can be disabled:

```
>>> stretch([-1., 0., 0.5, 1., 1.5], clip=False)
array([       nan, 0.        , 0.70710678, 1.        , 1.22474487])
```

Note

The stretch functions are similar but not always strictly identical to those used in e.g. DS9 (although they should have the same behavior). The equations for the DS9 stretches can be found here and can be compared to the equations for our stretches provided in the `astropy.visualization` API section. The main difference between our stretches and DS9 is that we have adjusted them so that the [0:1] range always maps exactly to the [0:1] range.

Combining transformations

Any intervals and stretches can be chained by using the `+` operator, which returns a new transformation. When combining intervals and stretches, the stretch object must come before the interval object. For example, to apply normalization based on a percentile value, followed by a square root stretch, you can do:

```
>>> transform = SqrtStretch() + PercentileInterval(90.)
>>> transform([1, 3, 4, 5, 6])
array([0.        , 0.60302269, 0.76870611, 0.90453403, 1.        ])
```

As before, the combined transformation can also accept a `clip` argument (which is `True` by default).

Matplotlib normalization

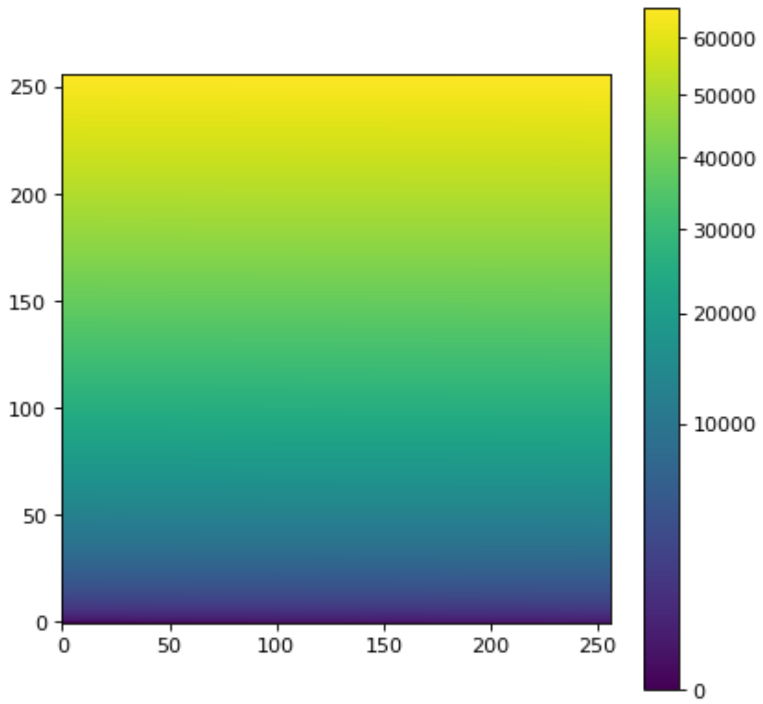
Matplotlib allows a custom normalization and stretch to be used when displaying data by passing a `matplotlib.colors.Normalize` object, e.g. to `imshow()`. The `astropy.visualization` module provides an `ImageNormalize` class that wraps the interval (see Intervals and

Normalization) and stretch (see Stretching) objects into an object Matplotlib understands.

The inputs to the **ImageNormalize** class are the data and the interval and stretch objects:

```
import numpy as np
import matplotlib.pyplot as plt

from astropy.visualization import (MinMaxInterval, SqrtStretch,
                                   ImageNormalize)

# Generate a test image
image = np.arange(65536).reshape((256, 256))

# Create an ImageNormalize object
norm = ImageNormalize(image, interval=MinMaxInterval(),
                      stretch=SqrtStretch())

# or equivalently using positional arguments
# norm = ImageNormalize(image, MinMaxInterval(), SqrtStretch())

# Display the image
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
im = ax.imshow(image, origin='lower', norm=norm)
fig.colorbar(im)
```

(png, svg, pdf)



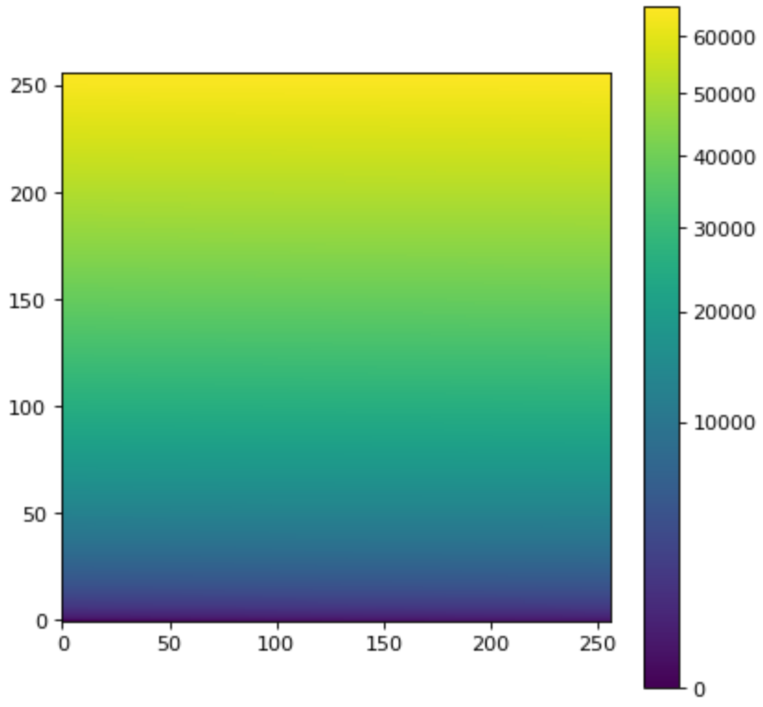
As shown above, the colorbar ticks are automatically adjusted.

Please note that one should not use `ax.imshow(norm(image))` because the colorbar ticks marks will represent normalized image values (on a linear scale), not the actual image values. Also, the image displayed by `ax.imshow(norm(image))` is not exactly equivalent to `ax.imshow(image, norm=norm)` if the image contains `NaN` or `inf` values. The exact equivalent is `ax.imshow(norm(np.ma.masked_invalid(image))`.

The input image to **`ImageNormalize`** is typically the one to be displayed, so there is a convenience function **`imshow_norm()`** to ease this use case:

```
import numpy as np
import matplotlib.pyplot as plt

from astropy.visualization import imshow_norm, MinMaxInterval,
SqrtStretch

# Generate a test image
image = np.arange(65536).reshape((256, 256))

# Display the exact same thing as the above plot
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
im, norm = imshow_norm(image, ax, origin='lower',
                       interval=MinMaxInterval(),
stretch=SqrtStretch())
fig.colorbar(im)
```

(png, svg, pdf)

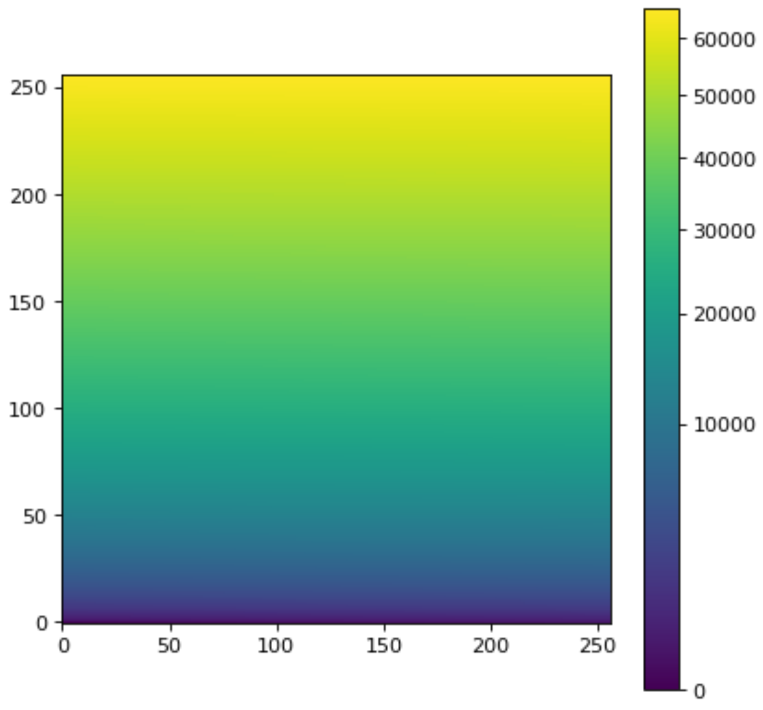


While this is the simplest case, it is also possible for a completely different image to be used to establish the normalization (e.g. if one wants to display several images with exactly the same normalization and stretch).

The inputs to the **`ImageNormalize`** class can also be the vmin and vmax limits, which you can determine from the Intervals and Normalization classes, and the stretch object:

```python
import numpy as np
import matplotlib.pyplot as plt

from astropy.visualization import (MinMaxInterval, SqrtStretch,
                                   ImageNormalize)

# Generate a test image
image = np.arange(65536).reshape((256, 256))

# Create interval object
interval = MinMaxInterval()
vmin, vmax = interval.get_limits(image)

# Create an ImageNormalize object using a SqrtStretch object
norm = ImageNormalize(vmin=vmin, vmax=vmax, stretch=SqrtStretch())

# Display the image
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
im = ax.imshow(image, origin='lower', norm=norm)
fig.colorbar(im)
```

(png, svg, pdf)

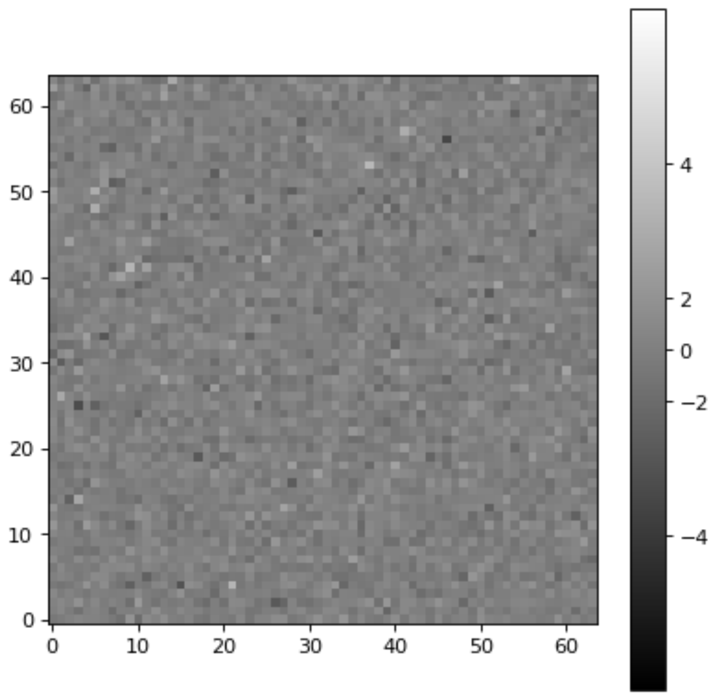


Combining stretches and Matplotlib normalization

Stretches can also be combined with other stretches, just like transformations. The resulting **CompositeStretch** can be used to normalize Matplotlib images like any other stretch. For example, a composite stretch can stretch residual images with negative values:

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.visualization.stretch import SinhStretch, LinearStretch
from astropy.visualization import ImageNormalize

# Transforms normalized values [0,1] to [-1,1] before stretch and
then back
stretch = LinearStretch(slope=0.5, intercept=0.5) + SinhStretch() + \
    LinearStretch(slope=2, intercept=-1)

# Image of random Gaussian noise
image = np.random.normal(size=(64, 64))
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
# ImageNormalize normalizes values to [0,1] before applying the
stretch
norm = ImageNormalize(stretch=stretch, vmin=-5, vmax=5)
im = ax.imshow(image, origin='lower', norm=norm, cmap='gray')
fig.colorbar(im)
```

(png, svg, pdf)



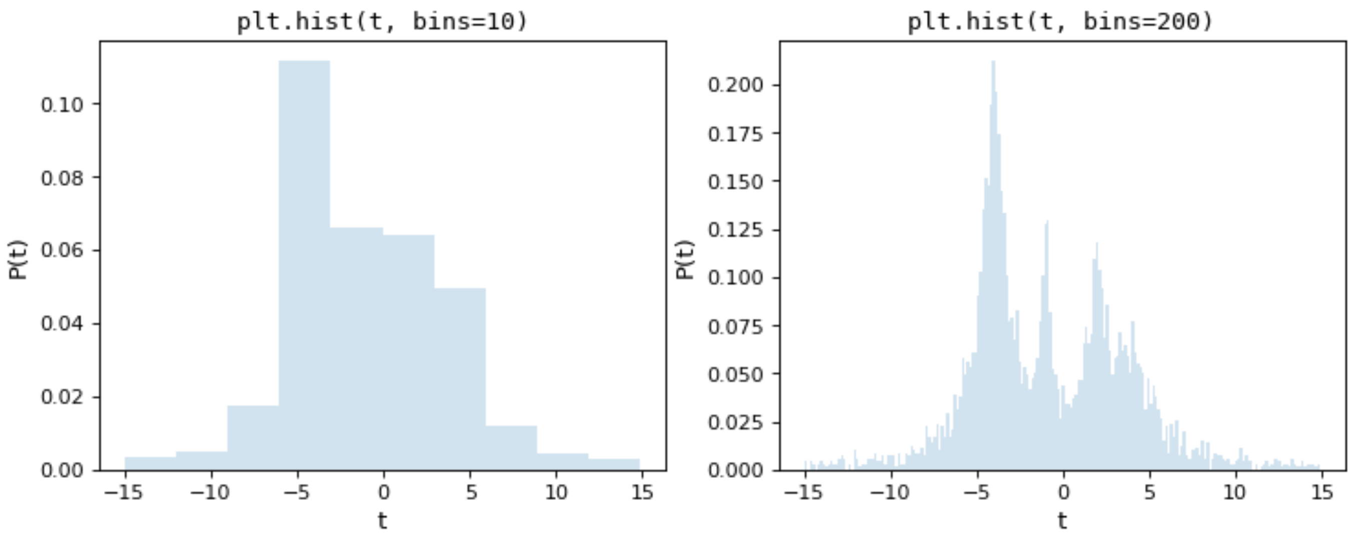
Choosing Histogram Bins

The **`astropy.visualization`** module provides the **`hist()`** function, which is a generalization of matplotlib's histogram function which allows for more flexible specification of histogram bins. For computing bins without the accompanying plot, see **`astropy.stats.histogram()`**.

As a motivation for this, consider the following two histograms, which are constructed from the same underlying set of 5000 points, the first with matplotlib's default of 10 bins, the second with an arbitrarily chosen 200 bins:

```
import numpy as np
import matplotlib.pyplot as plt

# generate some complicated data
rng = np.random.RandomState(0)
t = np.concatenate([-5 + 1.8 * rng.standard_cauchy(500),
                    -4 + 0.8 * rng.standard_cauchy(2000),
                    -1 + 0.3 * rng.standard_cauchy(500),
                    2 + 0.8 * rng.standard_cauchy(1000),
                    4 + 1.5 * rng.standard_cauchy(1000)])

# truncate to a reasonable range
t = t[(t > -15) & (t < 15)]

# draw histograms with two different bin widths
fig, ax = plt.subplots(1, 2, figsize=(10, 4))

fig.subplots_adjust(left=0.1, right=0.95, bottom=0.15)
for i, bins in enumerate([10, 200]):
    ax[i].hist(t, bins=bins, histtype='stepfilled', alpha=0.2,
density=True)
    ax[i].set_xlabel('t')
    ax[i].set_ylabel('P(t)')
    ax[i].set_title('plt.hist(t, bins={0})'.format(bins),
                    fontdict=dict(family='monospace'))
```

(png, svg, pdf)



Upon visual inspection, it is clear that each of these choices is suboptimal: with 10 bins, the fine structure of the data distribution is lost, while with 200 bins, heights of individual bins are affected by sampling error. The tried-and-true method employed by most scientists is a trial and error approach that attempts to find a suitable midpoint between these.

Astropy's **`hist()`** function addresses this by providing several methods of automatically tuning the histogram bin size. It has a syntax identical to matplotlib's `plt.hist` function, with the exception of the `bins` parameter, which allows specification of one of four different methods for automatic bin selection. These methods are implemented in **`astropy.stats.histogram()`**, which has a similar syntax to the `np.histogram` function.

Normal Reference Rules

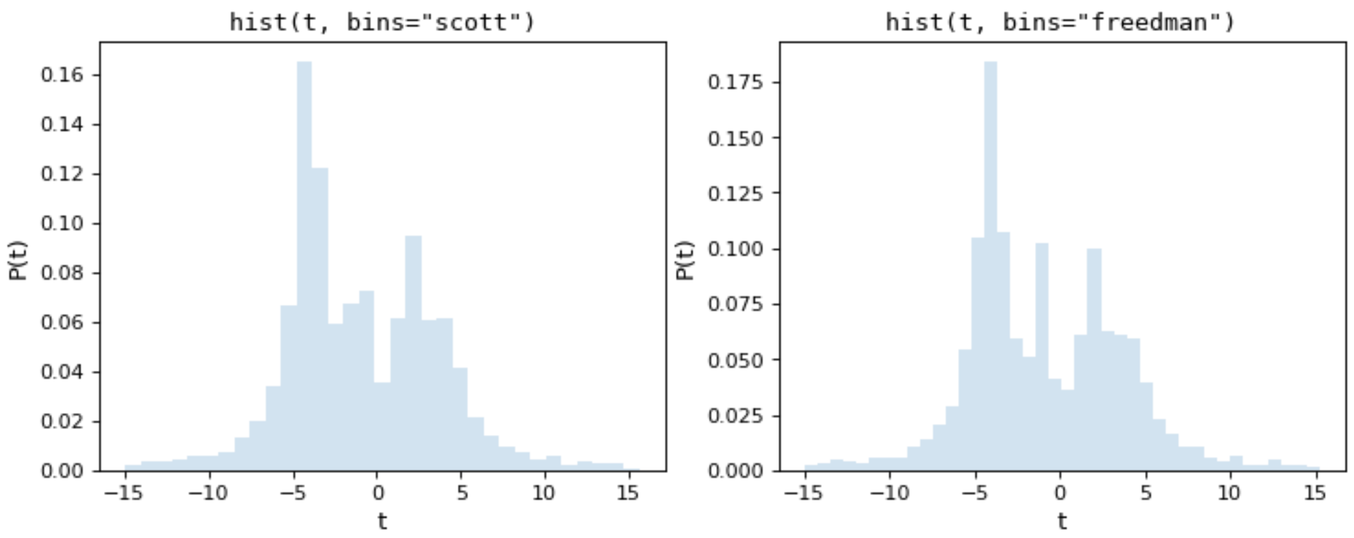
The simplest methods of tuning the number of bins are the normal reference rules due to Scott (implemented in **`scott_bin_width()`**) and Freedman & Diaconis (implemented in **`freedman_bin_width()`**). These rules proceed by assuming the data is close to normally-distributed, and applying a rule-of-thumb intended to minimize the difference between the histogram and the underlying distribution of data.

The following figure shows the results of these two rules on the above dataset:

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.visualization import hist
```

```
# generate some complicated data
rng = np.random.RandomState(0)
t = np.concatenate([-5 + 1.8 * rng.standard_cauchy(500),
                    -4 + 0.8 * rng.standard_cauchy(2000),
                    -1 + 0.3 * rng.standard_cauchy(500),
                    2 + 0.8 * rng.standard_cauchy(1000),
                    4 + 1.5 * rng.standard_cauchy(1000)])

# truncate to a reasonable range
t = t[(t > -15) & (t < 15)]

# draw histograms with two different bin widths
fig, ax = plt.subplots(1, 2, figsize=(10, 4))

fig.subplots_adjust(left=0.1, right=0.95, bottom=0.15)
for i, bins in enumerate(['scott', 'freedman']):
    hist(t, bins=bins, ax=ax[i], histtype='stepfilled',
         alpha=0.2, density=True)
    ax[i].set_xlabel('t')
    ax[i].set_ylabel('P(t)')
    ax[i].set_title('hist(t, bins="{0}")'.format(bins),
                    fontdict=dict(family='monospace'))
```

(png, svg, pdf)



As we can see, both of these rules of thumb choose an intermediate number of bins which provide a good trade-off between data representation and noise suppression.

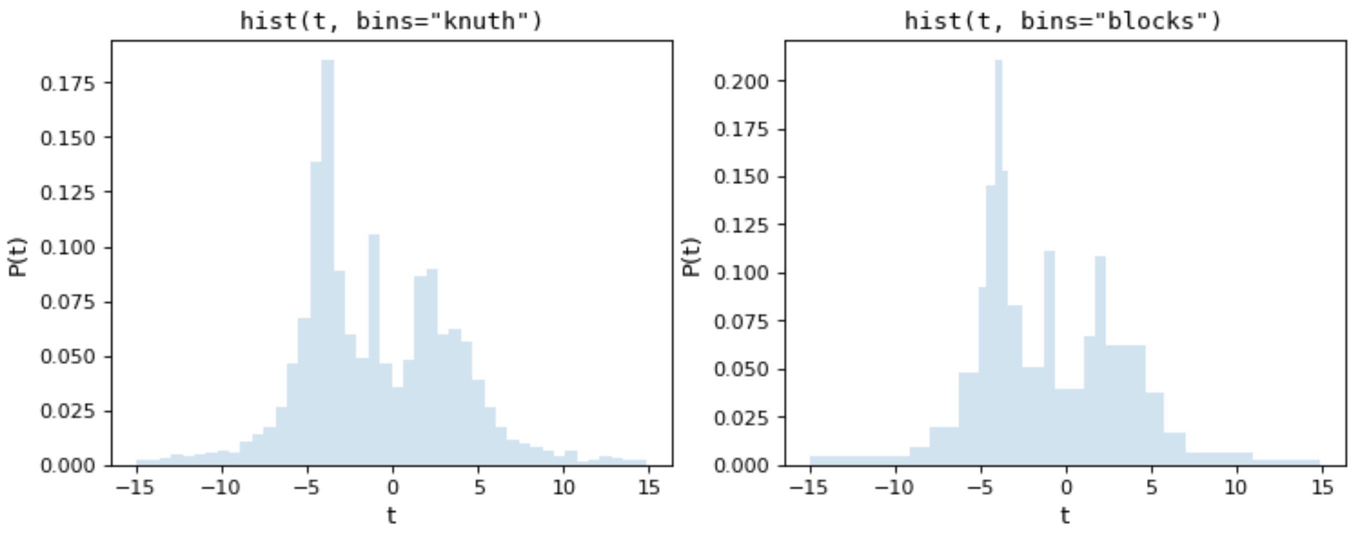
Bayesian Models

Though rules-of-thumb like Scott’s rule and the Freedman-Diaconis rule are fast and convenient, their strong assumptions about the data make them suboptimal for more complicated distributions. Other methods of bin selection use fitness functions computed on the actual data to choose an optimal binning. Astropy implements two of these examples: Knuth’s rule (implemented in **knuth_bin_width()**) and Bayesian Blocks (implemented in **bayesian_blocks()**).

Knuth’s rule chooses a constant bin size which minimizes the error of the histogram’s approximation to the data, while the Bayesian Blocks uses a more flexible method which allows varying bin widths. Because both of these require the minimization of a cost function across the dataset, they are more computationally intensive than the rules-of-thumb mentioned above. Here are the results of these procedures for the above dataset:

```python
import warnings
import numpy as np
import matplotlib.pyplot as plt
from astropy.visualization import hist

# generate some complicated data
rng = np.random.RandomState(0)
t = np.concatenate([-5 + 1.8 * rng.standard_cauchy(500),
                    -4 + 0.8 * rng.standard_cauchy(2000),
                    -1 + 0.3 * rng.standard_cauchy(500),
                    2 + 0.8 * rng.standard_cauchy(1000),
                    4 + 1.5 * rng.standard_cauchy(1000)])

# truncate to a reasonable range
t = t[(t > -15) & (t < 15)]

# draw histograms with two different bin widths
fig, ax = plt.subplots(1, 2, figsize=(10, 4))

fig.subplots_adjust(left=0.1, right=0.95, bottom=0.15)
for i, bins in enumerate(['knuth', 'blocks']):
    hist(t, bins=bins, ax=ax[i], histtype='stepfilled',
            alpha=0.2, density=True)
    ax[i].set_xlabel('t')
    ax[i].set_ylabel('P(t)')
    ax[i].set_title('hist(t, bins="{0}")'.format(bins),
                    fontdict=dict(family='monospace'))
```

(png, svg, pdf)



Notice that both of these capture the shape of the distribution very accurately, and that the `bins='blocks'` panel selects bin widths which vary in width depending on the local structure in the data. Compared to standard defaults, these Bayesian optimization methods provide a much more principled means of choosing histogram binning.

Creating color RGB images

RGB images can be produced using matplotlib's ability to make three-color images. In general, an RGB image is an MxNx3 array, where M is the y-dimension, N is the x-dimension, and the length-3 layer represents red, green, and blue, respectively. A fourth layer representing the alpha (opacity) value can be specified.

Matplotlib has several tools for manipulating these colors at **`matplotlib.colors`**.

Astropy's visualization tools can be used to change the stretch and scaling of the individual layers of the RGB image. Each layer must be on a scale of 0-1 for floats (or 0-255 for integers); values outside that range will be clipped.

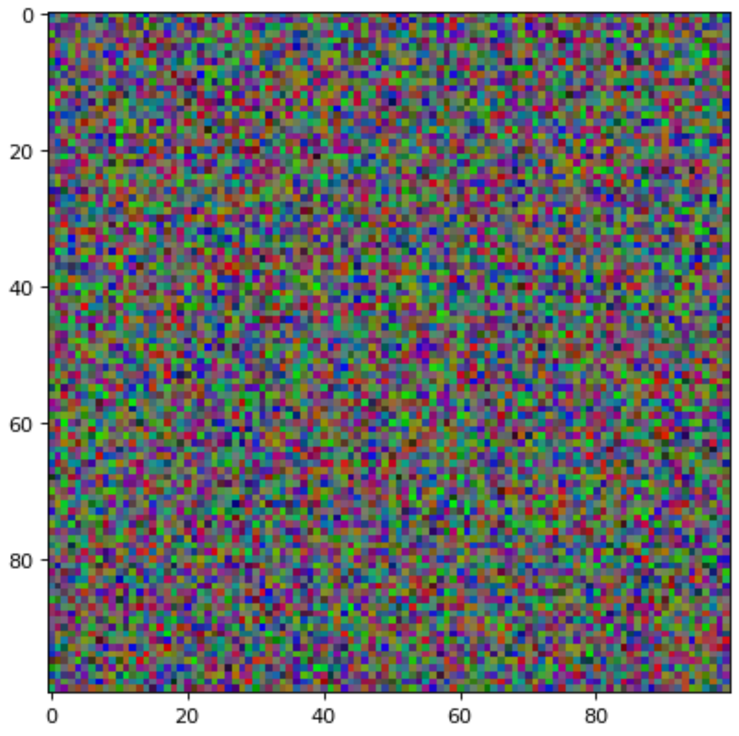
Creating color RGB images using the Lupton et al (2004) scheme

Lupton et al. (2004) describe an "optimal" algorithm for producing red-green-blue composite images from three separate high-dynamic range arrays. This method is implemented in **`make_lupton_rgb`** as a convenience wrapper function and an associated set of classes to provide alternate scalings. The SDSS SkyServer color images were made using a variation on this technique. To generate a color PNG file with the default (arcsinh) scaling:

```
import numpy as np
import matplotlib.pyplot as plt
from astropy.visualization import make_lupton_rgb
```

```
image_r = np.random.random((100,100))
image_g = np.random.random((100,100))
image_b = np.random.random((100,100))
image = make_lupton_rgb(image_r, image_g, image_b, stretch=0.5)
plt.imshow(image)
```

(png, svg, pdf)



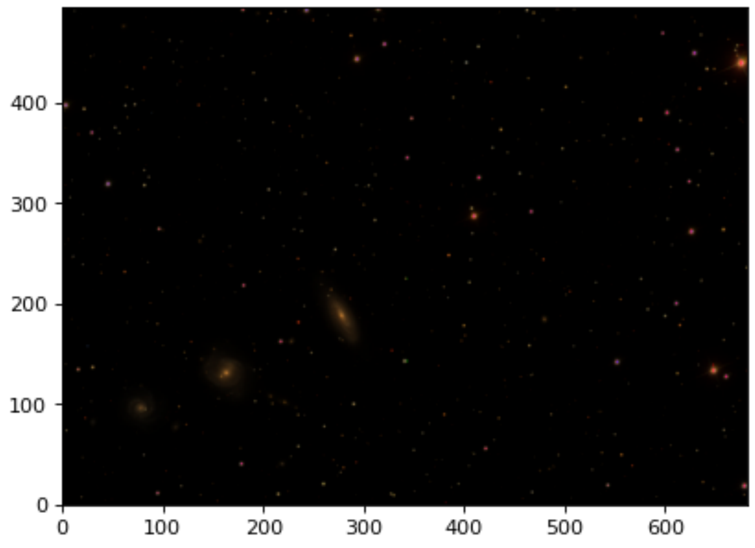
This method requires that the three images be aligned and have the same pixel scale and size. Changing `minimum` will change the black level, while `stretch` and `Q` will change how the values between black and white are scaled.

For a more in-depth example, download the `g`, `r`, `i` SDSS frames (they will serve as the blue, green and red channels respectively) of the area around the Hickson 88 group and try the example below and compare it with Figure 1 of Lupton et al. (2004):

```
import matplotlib.pyplot as plt
from astropy.visualization import make_lupton_rgb
from astropy.io import fits
from astropy.utils.data import get_pkg_data_filename

# Read in the three images downloaded from here:
g_name =
get_pkg_data_filename('visualization/reprojected_sdss_g.fits.bz2')
r_name =
get_pkg_data_filename('visualization/reprojected_sdss_r.fits.bz2')
```

```
i_name =
get_pkg_data_filename('visualization/reprojected_sdss_i.fits.bz2')
g = fits.open(g_name)[0].data
r = fits.open(r_name)[0].data
i = fits.open(i_name)[0].data

rgb_default = make_lupton_rgb(i, r, g, filename="ngc6976-
default.jpeg")
plt.imshow(rgb_default, origin='lower')
```

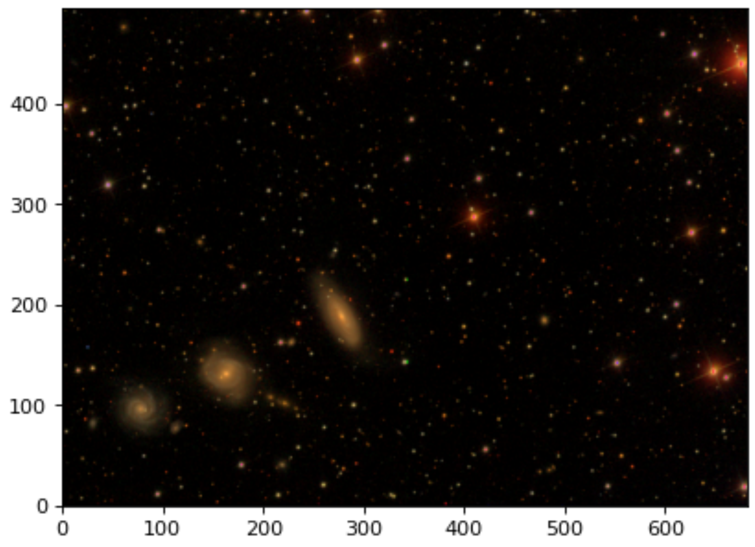
(png, svg, pdf)



The image above was generated with the default parameters. However using a different scaling, e.g Q=10, stretch=0.5, faint features of the galaxies show up. Compare with Fig. 1 of Lupton et al. (2004) or the SDSS Skyserver image.

```
rgb = make_lupton_rgb(i, r, g, Q=10, stretch=0.5,
filename="ngc6976.jpeg")
plt.imshow(rgb, origin='lower')
```

(png, svg, pdf)



Scripts

This module includes a command-line script, `fits2bitmap` to convert FITS images to bitmaps, including scaling and stretching of the image. To find out more about the available options and how to use it, type:

```
$ fits2bitmap --help
```

Reference/API

astropy.visualization Package

Functions

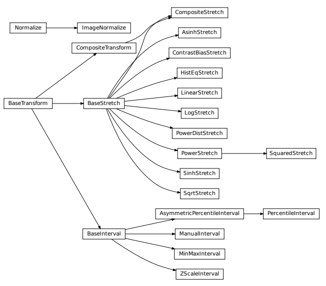
`hist`(x[, bins, ax, max_bins])	Enhanced histogram function
`imshow_norm`(data[, ax, imshow_only_kwargs])	A convenience function to call matplotlib's `matplotlib.pyplot.imshow` function, using an `ImageNormalize` object as the normalization.
`make_lupton_rgb`(image_r, image_g, image_b[, …])	Return a Red/Green/Blue color image from up to 3 images using an asinh stretch.
`quantity_support`([format])	Enable support for plotting `astropy.units.Quantity` instances in matplotlib.
`simple_norm`(data[, stretch, power, asinh_a, …])	Return a Normalization class that can be used for displaying images with Matplotlib.

`time_support`(*[, scale, format, simplify])	Enable support for plotting `astropy.time.Time` instances in matplotlib.

Classes

`AsinhStretch`([a])	An asinh stretch.
`AsymmetricPercentileInterval`(…[, n_samples])	Interval based on a keeping a specified fraction of pixels (can be asymmetric).
`BaseInterval`()	Base class for the interval classes, which, when called with an array of values, return an interval computed following different algorithms.
`BaseStretch`()	Base class for the stretch classes, which, when called with an array of values in the range [0:1], return an transformed array of values, also in the range [0:1].
`BaseTransform`()	A transformation object.
`CompositeStretch`(transform_1, transform_2)	A combination of two stretches.
`CompositeTransform`(transform_1, transform_2)	A combination of two transforms.
`ContrastBiasStretch`(contrast, bias)	A stretch that takes into account contrast and bias.
`HistEqStretch`(data[, values])	A histogram equalization stretch.
`ImageNormalize`([data, interval, vmin, vmax, …])	Normalization class to be used with Matplotlib.
`LinearStretch`([slope, intercept])	A linear stretch with a slope and offset.
`LogStretch`([a])	A log stretch.
`ManualInterval`([vmin, vmax])	Interval based on user-specified values.
`MinMaxInterval`()	Interval based on the minimum and maximum values in the data.
`PercentileInterval`(percentile[, n_samples])	Interval based on a keeping a specified fraction of pixels.
`PowerDistStretch`([a])	An alternative power stretch.
`PowerStretch`(a)	A power stretch.
`SinhStretch`([a])	A sinh stretch.
`SqrtStretch`()	A square root stretch.
`SquaredStretch`()	A convenience class for a power stretch of 2.
`ZScaleInterval`([nsamples, contrast, …])	Interval based on IRAF’s zscale.

Class Inheritance Diagram



astropy.visualization.mpl_normalize Module

Normalization class for Matplotlib that can be used to produce colorbars.

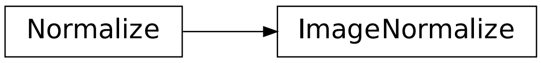
Functions

`simple_norm`(data[, stretch, power, asinh_a, …])	Return a Normalization class that can be used for displaying images with Matplotlib.
`imshow_norm`(data[, ax, imshow_only_kwargs])	A convenience function to call matplotlib's `matplotlib.pyplot.imshow` function, using an `ImageNormalize` object as the normalization.

Classes

`ImageNormalize`([data, interval, vmin, vmax, …])	Normalization class to be used with Matplotlib.

Class Inheritance Diagram



Astrostatistics Tools (`astropy.stats`)

Introduction

The **astropy.stats** package holds statistical functions or algorithms used in astronomy. While the **scipy.stats** and statsmodels packages contains a wide range of statistical tools, they are general-purpose packages and are missing some tools that are particularly useful or specific to astronomy. This package is intended to provide such functionality, but *not* to replace **scipy.stats** if its implementation satisfies astronomers' needs.

Getting Started

A number of different tools are contained in the stats package, and they can be accessed by importing them:

```
>>> from astropy import stats
```

A full list of the different tools are provided below. Please see the documentation for their different usages. For example, sigma clipping, which is a common way to estimate the background of an image, can be performed with the **sigma_clip()** function. By default, the function returns a masked array where outliers are masked.

Examples

To estimate the background of an image:

```
>>> data = [1, 5, 6, 8, 100, 5, 3, 2]
>>> stats.sigma_clip(data, sigma=2, maxiters=5)
masked_array(data=[1, 5, 6, 8, --, 5, 3, 2],
             mask=[False, False, False, False,  True, False, False,
False],
       fill_value=999999)
```

Alternatively, the **SigmaClip** class provides an object-oriented interface to sigma clipping, which also returns a masked array by default:

```
>>> sigclip = stats.SigmaClip(sigma=2, maxiters=5)
>>> sigclip(data)
masked_array(data=[1, 5, 6, 8, --, 5, 3, 2],
             mask=[False, False, False, False,  True, False, False,
False],
       fill_value=999999)
```

In addition, there are also several convenience functions for making the calculation of statistics even more convenient. For example, **sigma_clipped_stats()** will return the mean, median, and standard deviation of a sigma-clipped array:

```
>>> stats.sigma_clipped_stats(data, sigma=2, maxiters=5)
(4.2857142857142856, 5.0, 2.2497165354319457)
```

There are also tools for calculating robust statistics, sampling the data, circular statistics, confidence limits, spatial statistics, and adaptive histograms.

Most tools are fairly self-contained, and include relevant examples in their docstrings.

Using `astropy.stats`

More detailed information on using the package is provided on separate pages, listed below.

Robust Statistical Estimators

Robust statistics provides reliable estimates of basic statistics for complex distributions. The statistics package includes several robust statistical functions that are commonly used in astronomy. This includes methods for rejecting outliers as well as statistical description of the underlying distributions.

In addition to the functions mentioned here, models can be fit with outlier rejection using `FittingWithOutlierRemoval()`.

Sigma Clipping

Sigma clipping provides a fast method for identifying outliers in a distribution. For a distribution of points, a center and a standard deviation are calculated. Values which are less or more than a specified number of standard deviations from a center value are rejected. The process can be iterated to further reject outliers.

The `astropy.stats` package provides both a functional and object-oriented interface for sigma clipping. The function is called `sigma_clip()` and the class is called `SigmaClip`. By default, they both return a masked array where the rejected points are masked.

Examples

We can start by generating some data that has a mean of 0 and standard deviation of 0.2, but with outliers:

```
>>> import numpy as np
>>> import scipy.stats as stats
>>> np.random.seed(0)
>>> x = np.arange(200)
>>> y = np.zeros(200)
>>> c = stats.bernoulli.rvs(0.35, size=x.shape)
>>> y += (np.random.normal(0., 0.2, x.shape) +
...       c*np.random.normal(3.0, 5.0, x.shape))
```

Now we can use **sigma_clip()** to perform sigma clipping on the data:

```
>>> from astropy.stats import sigma_clip
>>> filtered_data = sigma_clip(y, sigma=3, maxiters=10)
```

The output masked array then can be used to calculate statistics on the data, fit models to the data, or otherwise explore the data.

To perform the same sigma clipping with the **SigmaClip** class:

```
>>> from astropy.stats import SigmaClip
>>> sigclip = SigmaClip(sigma=3, maxiters=10)
>>> print(sigclip)
<SigmaClip>
   sigma: 3
   sigma_lower: None
   sigma_upper: None
   maxiters: 10
   cenfunc: <function median at 0x108dbde18>
   stdfunc: <function std at 0x103ab52f0>
>>> filtered_data = sigclip(y)
```

Note that once the `sigclip` instance is defined above, it can be applied to other data using the same already defined sigma-clipping parameters.

For basic statistics, **sigma_clipped_stats()** is a convenience function to calculate the sigma-clipped mean, median, and standard deviation of an array. As can be seen, rejecting the outliers returns accurate values for the underlying distribution.

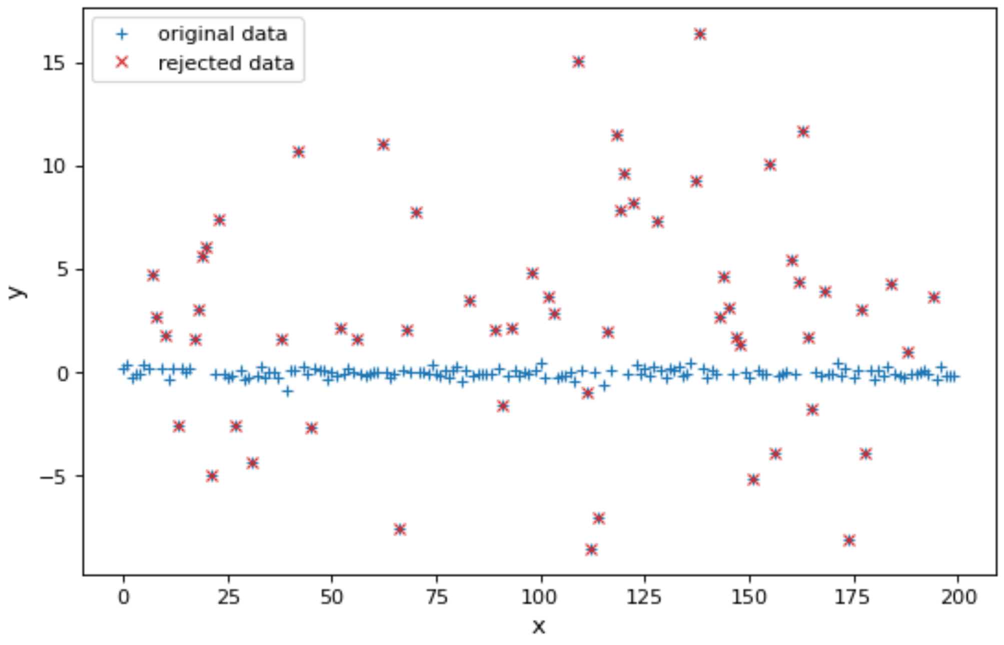
To use **sigma_clipped_stats()** for sigma-clipped statistics calculation:

```
>>> from astropy.stats import sigma_clipped_stats
>>> y.mean(), np.median(y), y.std()
(0.86586417693378226, 0.03265864495523732, 3.2913811977676444)
>>> sigma_clipped_stats(y, sigma=3, maxiters=10)
(-0.0020337793767186197, -0.023632809025713953, 0.19514652532636906)
```

sigma_clip() and **SigmaClip** can be combined with other robust statistics to provide improved outlier rejection as well.

```
import numpy as np
import scipy.stats as stats
from matplotlib import pyplot as plt
from astropy.stats import sigma_clip, mad_std

# Generate fake data that has a mean of 0 and standard deviation of
0.2 with outliers
np.random.seed(0)
x = np.arange(200)
y = np.zeros(200)
c = stats.bernoulli.rvs(0.35, size=x.shape)
y += (np.random.normal(0., 0.2, x.shape) +
      c*np.random.normal(3.0, 5.0, x.shape))

filtered_data = sigma_clip(y, sigma=3, maxiters=1, stdfunc=mad_std)

# plot the original and rejected data
plt.figure(figsize=(8,5))
plt.plot(x, y, '+', color='#1f77b4', label="original data")
plt.plot(x[filtered_data.mask], y[filtered_data.mask], 'x',
         color='#d62728', label="rejected data")
plt.xlabel('x')
plt.ylabel('y')
plt.legend(loc=2, numpoints=1)
```

(png, svg, pdf)



astropy.stats.sigma_clipping Module

Functions

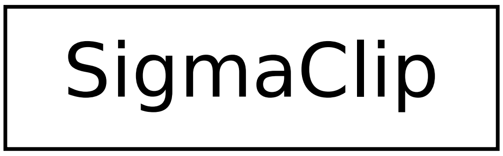
sigma_clip(data[, sigma, sigma_lower, …])	Perform sigma-clipping on the provided data.

sigma_clipped_stats(data[, mask, …])	Calculate sigma-clipped statistics on the provided data.

Classes

SigmaClip([sigma, sigma_lower, sigma_upper, …])	Class to perform sigma clipping.

Class Inheritance Diagram



Median Absolute Deviation

The median absolute deviation (MAD) is a measure of the spread of a distribution and is defined as `median(abs(a - median(a)))`. The MAD can be calculated using **median_absolute_deviation**. For a normal distribution, the MAD is related to the standard deviation by a factor of 1.4826, and a convenience function, **mad_std**, is available to apply the conversion.

Note

A function can be supplied to the **median_absolute_deviation** to specify the median function to be used in the calculation. Depending on the version of NumPy and whether the array is masked or contains irregular values, significant performance increases can be had by preselecting the median function. If the median function is not specified, **median_absolute_deviation** will attempt to select the most relevant function according to the input data.

Biweight Estimators

A set of functions are included in the **astropy.stats** package that use the biweight formalism. These functions have long been used in astronomy, particularly to calculate the velocity dispersion of galaxy clusters [1]. The following set of tasks are available for biweight measurements:

astropy.stats.biweight Module

This module contains functions for computing robust statistics using Tukey's biweight function.

Functions

biweight_location(data[, c, M, axis, ignore_nan])	Compute the biweight location.
biweight_scale(data[, c, M, axis, …])	Compute the biweight scale.
biweight_midvariance(data[, c, M, axis, …])	Compute the biweight midvariance.
biweight_midcovariance(data[, c, M, …])	Compute the biweight midcovariance between pairs of multiple variables.
biweight_midcorrelation(x, y[, c, M, …])	Compute the biweight midcorrelation between two variables.

References

[1] Beers, Flynn, and Gebhardt (1990; AJ 100, 32) (https://ui.adsabs.harvard.edu/abs/1990AJ….100…32B)

Circular Statistics

astropy.stats.circstats Module

This module contains simple functions for dealing with circular statistics, for instance, mean, variance, standard deviation, correlation coefficient, and so on. This module also cover tests of uniformity, e.g., the Rayleigh and V tests. The Maximum Likelihood Estimator for the Von Mises distribution along with the Cramer-Rao Lower Bounds are also implemented. Almost all of the implementations are based on reference [1], which is also the basis for the R package 'CircStats' [2].

Functions

circmean(data[, axis, weights])	Computes the circular mean angle of an array of circular data.
circstd(data[, axis, weights, method])	Computes the circular standard deviation of an array of circular data.
circvar(data[, axis, weights])	Computes the circular variance of an array of circular data.
circmoment(data[, p, centered, axis, weights])	Computes the `p` -th trigonometric circular moment for an array of circular data.
circcorrcoef(alpha, beta[, axis, …])	Computes the circular correlation coefficient between two array of circular data.
rayleightest(data[, axis, weights])	Performs the Rayleigh test of uniformity.

`vtest`(data[, mu, axis, weights])	Performs the Rayleigh test of uniformity where the alternative hypothesis H1 is assumed to have a known mean angle `mu`.
`vonmisesmle`(data[, axis])	Computes the Maximum Likelihood Estimator (MLE) for the parameters of the von Mises distribution.

References

[1] S. R. Jammalamadaka, A. SenGupta. “Topics in Circular Statistics”. Series on Multivariate Analysis, Vol. 5, 2001.

[2] C. Agostinelli, U. Lund. “Circular Statistics from ‘Topics in Circular Statistics (2001)’”. 2015.

Ripley’s K Function Estimators

Spatial correlation functions have been used in the astronomical context to estimate the probability of finding an object (e.g., a galaxy) within a given distance of another object [1].

Ripley’s K function is a type of estimator used to characterize the correlation of such spatial point processes [2], [3], [4], [5], [6]. More precisely, it describes correlation among objects in a given field. The **`RipleysKEstimator`** class implements some estimators for this function which provides several methods for edge effects correction.

Basic Usage

The actual implementation of Ripley’s K function estimators lie in the method `evaluate`, which take the following arguments: `data`, `radii`, and optionally, `mode`.

The `data` argument is a 2D array which represents the set of observed points (events) in the area of study. The `radii` argument corresponds to a set of distances for which the estimator will be evaluated. The `mode` argument takes a value on the following linguistic set `{none, translation, ohser, var-width, ripley}`; each keyword represents a different method to perform correction due to edge effects. See the API documentation and references for details about these methods.

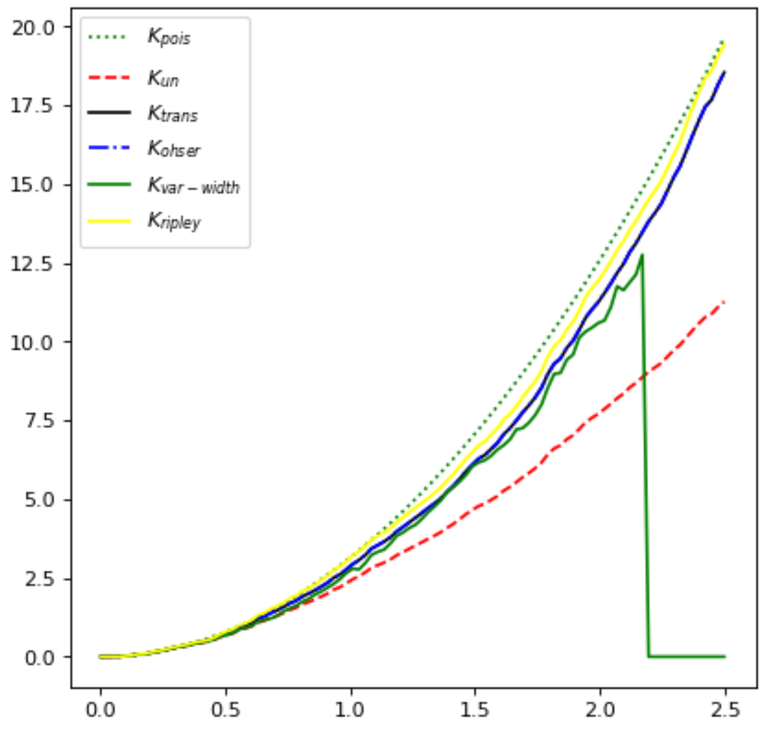
Instances of **`RipleysKEstimator`** can also be used as callables (which is equivalent to calling the `evaluate` method).

Example

To use Ripley's K Function Estimators from `astropy`'s stats sub-package:

```
import numpy as np
from matplotlib import pyplot as plt
from astropy.stats import RipleysKEstimator

z = np.random.uniform(low=5, high=10, size=(100, 2))
Kest = RipleysKEstimator(area=25, x_max=10, y_max=10, x_min=5,
y_min=5)

r = np.linspace(0, 2.5, 100)
plt.plot(r, Kest.poisson(r), color='green', ls=':',
label=r'$K_{pois}$')
plt.plot(r, Kest(data=z, radii=r, mode='none'), color='red', ls='--',
         label=r'$K_{un}$')
plt.plot(r, Kest(data=z, radii=r, mode='translation'), color='black',
         label=r'$K_{trans}$')
plt.plot(r, Kest(data=z, radii=r, mode='ohser'), color='blue',
ls='-.',
         label=r'$K_{ohser}$')
plt.plot(r, Kest(data=z, radii=r, mode='var-width'), color='green',
         label=r'$K_{var-width}$')
plt.plot(r, Kest(data=z, radii=r, mode='ripley'), color='yellow',
         label=r'$K_{ripley}$')
plt.legend()
```

(png, svg, pdf)



References

[1] Peebles, P.J.E. *The large scale structure of the universe*. <https://ui.adsabs.harvard.edu/abs/1980lssu.book…..P>

[2] Ripley, B.D. *The second-order analysis of stationary point processes*. Journal of Applied Probability. 13: 255–266, 1976.

[3] *Spatial descriptive statistics*. <https://en.wikipedia.org/wiki/Spatial_descriptive_statistics>

[4] Cressie, N.A.C. *Statistics for Spatial Data*, Wiley, New York.

[5] Stoyan, D., Stoyan, H. *Fractals, Random Shapes and Point Fields*, Akademie Verlag GmbH, Chichester, 1992.

[6] *Correlation function*. <https://en.wikipedia.org/wiki/Correlation_function_(astronomy)>

Constants

The `astropy.stats` package defines two constants useful for converting between Gaussian sigma and full width at half maximum (FWHM):

`gaussian_sigma_to_fwhm`

Factor with which to multiply Gaussian 1-sigma standard deviation to convert it to full width at half maximum (FWHM).

```
>>> from astropy.stats import gaussian_sigma_to_fwhm
>>> gaussian_sigma_to_fwhm
2.3548200450309493
```

`gaussian_fwhm_to_sigma`

Factor with which to multiply Gaussian full width at half maximum (FWHM) to convert it to 1-sigma standard deviation.

```
>>> from astropy.stats import gaussian_fwhm_to_sigma
>>> gaussian_fwhm_to_sigma
0.42466090014400953
```

See Also

- `scipy.stats`
  This SciPy package contains a variety of useful statistical functions and classes. The functionality in `astropy.stats` is intended to supplement this, *not* replace it.

- statsmodels

  The statsmodels package provides functionality for estimating different statistical models, tests, and data exploration.

- astroML

  The astroML package is a Python module for machine learning and data mining. Some of the tools from this package have been migrated here, but there are still a number of tools there that are useful for astronomy and statistical analysis.

- **`astropy.visualization.hist()`**

  The **`histogram()`** routine and related functionality defined here are used within the **`astropy.visualization.hist()`** function. For a discussion of these methods for determining histogram binnings, see Choosing Histogram Bins.

Performance Tips

If you are finding sigma clipping to be slow, and if you have not already done so, consider installing the bottleneck package, which will speed up some of the internal computations. In addition, if you are using standard functions for `cenfunc` and/or `stdfunc`, make sure you specify these as strings rather than passing a NumPy function — that is, use:

```
>>> sigma_clip(array, cenfunc='median')
```

instead of:

```
>>> sigma_clip(array, cenfunc=np.nanmedian)
```

Using strings will allow the sigma-clipping algorithm to pick the fastest implementation available for finding the median.

Reference/API

astropy.stats Package

This subpackage contains statistical tools provided for or used by Astropy.

While the **`scipy.stats`** package contains a wide range of statistical tools, it is a general-purpose package, and is missing some that are particularly useful to astronomy or are used in an atypical way in astronomy. This package is intended to provide such functionality, but *not* to replace **`scipy.stats`** if its implementation satisfies astronomers’ needs.

Functions

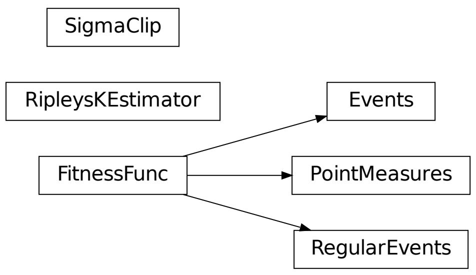
`binom_conf_interval`(k, n[, …])	Binomial proportion confidence interval given k successes, n trials.
`binned_binom_proportion`(x, success[, bins, …])	Binomial proportion and confidence interval in bins of a continuous variable `x`.
`poisson_conf_interval`(n[, interval, sigma, …])	Poisson parameter confidence interval given observed counts
`median_absolute_deviation`(data[, axis, …])	Calculate the median absolute deviation (MAD).
`mad_std`(data[, axis, func, ignore_nan])	Calculate a robust standard deviation using the median absolute deviation (MAD).
`signal_to_noise_oir_ccd`(t, source_eps, …)	Computes the signal to noise ratio for source being observed in the optical/IR using a CCD.
`bootstrap`(data[, bootnum, samples, bootfunc])	Performs bootstrap resampling on numpy arrays.
`kuiper`(data[, cdf, args])	Compute the Kuiper statistic.
`kuiper_two`(data1, data2)	Compute the Kuiper statistic to compare two samples.
`kuiper_false_positive_probability`(D, N)	Compute the false positive probability for the Kuiper statistic.
`cdf_from_intervals`(breaks, totals)	Construct a callable piecewise-linear CDF from a pair of arrays.
`interval_overlap_length`(i1, i2)	Compute the length of overlap of two intervals.
`histogram_intervals`(n, breaks, totals)	Histogram of a piecewise-constant weight function.
`fold_intervals`(intervals)	Fold the weighted intervals to the interval (0,1).
`biweight_location`(data[, c, M, axis, ignore_nan])	Compute the biweight location.
`biweight_scale`(data[, c, M, axis, …])	Compute the biweight scale.
`biweight_midvariance`(data[, c, M, axis, …])	Compute the biweight midvariance.
`biweight_midcovariance`(data[, c, M, …])	Compute the biweight midcovariance between pairs of multiple variables.
`biweight_midcorrelation`(x, y[, c, M, …])	Compute the biweight midcorrelation between two variables.
`sigma_clip`(data[, sigma, sigma_lower, …])	Perform sigma-clipping on the provided data.
`sigma_clipped_stats`(data[, mask, …])	Calculate sigma-clipped statistics on the provided data.
`jackknife_resampling`(data)	Performs jackknife resampling on numpy arrays.
`jackknife_stats`(data, statistic[, …])	Performs jackknife estimation on the basis of jackknife resamples.
`circmean`(data[, axis, weights])	Computes the circular mean angle of an array of circular data.
`circstd`(data[, axis, weights, method])	Computes the circular standard deviation of an array of circular data.
`circvar`(data[, axis, weights])	Computes the circular variance of an array of circular data.
`circmoment`(data[, p, centered, axis, weights])	Computes the `p` -th trigonometric circular moment for an array of circular data.

`circcorrcoef`(alpha, beta[, axis, …])	Computes the circular correlation coefficient between two array of circular data.
`rayleightest`(data[, axis, weights])	Performs the Rayleigh test of uniformity.
`vtest`(data[, mu, axis, weights])	Performs the Rayleigh test of uniformity where the alternative hypothesis H1 is assumed to have a known mean angle `mu`.
`vonmisesmle`(data[, axis])	Computes the Maximum Likelihood Estimator (MLE) for the parameters of the von Mises distribution.
`bayesian_blocks`(t[, x, sigma, fitness])	Compute optimal segmentation of data with Scargle's Bayesian Blocks
`histogram`(a[, bins, range, weights])	Enhanced histogram function, providing adaptive binnings
`scott_bin_width`(data[, return_bins])	Return the optimal histogram bin width using Scott's rule
`freedman_bin_width`(data[, return_bins])	Return the optimal histogram bin width using the Freedman-Diaconis rule
`knuth_bin_width`(data[, return_bins, quiet])	Return the optimal histogram bin width using Knuth's rule.
`calculate_bin_edges`(a[, bins, range, weights])	Calculate histogram bin edges like `numpy.histogram_bin_edges`.
`bayesian_info_criterion`(log_likelihood, …)	Computes the Bayesian Information Criterion (BIC) given the log of the likelihood function evaluated at the estimated (or analytically derived) parameters, the number of parameters, and the number of samples.
`bayesian_info_criterion_lsq`(ssr, n_params, …)	Computes the Bayesian Information Criterion (BIC) assuming that the observations come from a Gaussian distribution.
`akaike_info_criterion`(log_likelihood, …)	Computes the Akaike Information Criterion (AIC).
`akaike_info_criterion_lsq`(ssr, n_params, …)	Computes the Akaike Information Criterion assuming that the observations are Gaussian distributed.

Classes

`SigmaClip`([sigma, sigma_lower, sigma_upper, …])	Class to perform sigma clipping.
`FitnessFunc`([p0, gamma, ncp_prior])	Base class for bayesian blocks fitness functions
`Events`([p0, gamma, ncp_prior])	Bayesian blocks fitness for binned or unbinned events
`RegularEvents`(dt[, p0, gamma, ncp_prior])	Bayesian blocks fitness for regular events
`PointMeasures`([p0, gamma, ncp_prior])	Bayesian blocks fitness for point measures
`RipleysKEstimator`(area[, x_max, y_max, …])	Estimators for Ripley's K function for two-dimensional spatial data.

Class Inheritance Diagram



Nuts and bolts

Configuration System (astropy.config)

Introduction

The Astropy configuration system is designed to give users control of various parameters used in Astropy or affiliated packages without delving into the source code to make those changes.

Note

The configuration system got a major overhaul in `astropy` 0.4 as part of APE3. See Configuration Transition for information about updating code to use the new API.

Getting Started

The Astropy configuration options are most conveniently set by modifying the configuration file. It will be automatically generated with all of the default values commented out the first time you import Astropy. You can find the exact location by doing:

```
>>> from astropy.config import get_config_dir
>>> get_config_dir()
```

And you should see the location of your configuration directory. The standard scheme generally puts your configuration directory in `$HOME/.astropy/config`. It can be customized with the environment variable `XDG_CONFIG_HOME` and the `$XDG_CONFIG_HOME/astropy` directory must exist. Note that `XDG_CONFIG_HOME` comes from a Linux-centric specification (see here for more details), but Astropy will use this on any OS as a more general means to know where user-specific configurations should be written.

Note

See Astropy's Default Configuration File for the content of this configuration file.

Once you have found the configuration file, open it with your favorite editor. It should have all of the sections you might want, with descriptions and the type of value that is accepted. Feel free to edit this as you wish, and any of these changes will be reflected when you next start Astropy. Or, if you want to see your changes immediately in your current Astropy session, do:

```
>>> from astropy.config import reload_config
>>> reload_config()
```

Note

If for whatever reason your `$HOME/.astropy` directory is not accessible (i.e., you have `astropy` running somehow as root but you are not the root user), the best solution is to set the `XDG_CONFIG_HOME` and `XDG_CACHE_HOME` environment variables pointing to directories, and create an `astropy` directory inside each of those. Both the configuration and data download systems will then use those directories and never try to access the `$HOME/.astropy` directory.

Using `astropy.config`

Accessing Values

By convention, configuration parameters live inside of objects called `conf` at the root of each sub-package. For example, configuration parameters related to data files live in `astropy.utils.data.conf`. This object has properties for getting and setting individual configuration parameters. For instance, to get the default URL for `astropy` remote data, do:

```
>>> from astropy.utils.data import conf
>>> conf.dataurl
'http://data.astropy.org/'
```

Changing Values at Runtime

Changing the persistent state of configuration values is done by editing the configuration file as described above. Values can also, however, be modified in an active Python session by setting any of the properties on a `conf` object.

Example

If there is a part of your configuration file that looks like:

```
[utils.data]

# URL for astropy remote data site.
dataurl = http://data.astropy.org/

# Time to wait for remote data query (in seconds).
remote_timeout = 3.0
```

You should be able to modify the values at runtime this way:

```
>>> from astropy.utils.data import conf
>>> conf.dataurl
'http://data.astropy.org/'
>>> conf.dataurl = 'http://astropydata.mywebsite.com'
>>> conf.dataurl
'http://astropydata.mywebsite.com'
>>> conf.remote_timeout
3.0
>>> conf.remote_timeout = 4.5
>>> conf.remote_timeout
4.5
```

Reloading Configuration

Instead of modifying the variables in Python, you can also modify the configuration files and then reload them.

Example

If you modify the configuration file to say:

```
[utils.data]

# URL for astropy remote data site.
dataurl = http://myotherdata.mywebsite.com/

# Time to wait for remote data query (in seconds).
remote_timeout = 6.3
```

And then run the following commands:

```
>>> conf.reload('dataurl')
>>> conf.reload('remote_timeout')
```

This should update the variables with the values from the configuration file:

```
>>> conf.dataurl
'http://myotherdata.mywebsite.com/'
>>> conf.remote_timeout
6.3
```

You can reload all configuration parameters of a `conf` object at once by calling `reload` with no parameters:

```
>>> conf.reload()
```

Or if you want to reload all Astropy configuration at once, use the **`reload_config`** function:

```
>>> from astropy import config
>>> config.reload_config('astropy')
```

You can also reset a configuration parameter back to its default value. Note that this is the default value defined in the Python code, and has nothing to do with the configuration file on disk:

```
>>> conf.reset('dataurl')
>>> conf.dataurl
'http://data.astropy.org/'
```

Exploring Configuration

To see what configuration parameters are defined for a given `conf`:

```
>>> from astropy.utils.iers import conf
>>> [key for key in conf]
['auto_download',
 'auto_max_age',
 ...,
 'ietf_leap_second_auto_url']
>>> conf.auto_max_age
30.0
```

You can also iterate through `conf` in a dictionary-like fashion:

```
>>> [key for key in conf.keys()]
```

```
['auto_download',
 'auto_max_age',
 ...,
 'ietf_leap_second_auto_url']
>>> [cfgitem for cfgitem in conf.values()]
[<ConfigItem: name='auto_download' value=True at ...>,
 <ConfigItem: name='auto_max_age' value=30.0 at ...>,
 ...,
 <ConfigItem: name='ietf_leap_second_auto_url' value=...>]
>>> for (key, cfgitem) in conf.items():
...     if key == 'auto_max_age':
...         print(f'{cfgitem.description} Value is {cfgitem()}')
Maximum age (days) of predictive data before auto-downloading.
Default is 30. Value is 30.0
```

Upgrading `astropy`

Each time you upgrade to a new major version of `astropy`, the configuration parameters may have changed.

If you never edited your configuration file, there is nothing for you to do. It will automatically be replaced with a configuration file template for the newly installed version of `astropy`.

If you did customize your configuration file, it will not be touched. Instead, a new configuration file template will be installed alongside it with the version number in the filename, for example `astropy.0.4.cfg`. You can compare this file to your `astropy.cfg` file to see what needs to be changed or updated.

Adding New Configuration Items

Configuration items should be used wherever an option or setting is needed that is either tied to a system configuration or should persist across sessions of `astropy` or an affiliated package. Options that may affect the results of science calculations should not be configuration items, but should instead be **`astropy.utils.state.ScienceState`**, so it is possible to reproduce science results without them being affected by configuration parameters set in a particular environment. Admittedly, this is only a guideline, as the precise cases where a configuration item is preferred over, say, a keyword option for a function is somewhat personal preference. It is the preferred form of persistent configuration, however, and `astropy` packages must all use it (and it is recommended for affiliated packages).

The reference guide below describes the interface for creating a `conf` object with a number of configuration parameters. They should be defined at the top level (i.e., in the `__init__.py` of each sub-package that has configuration items):

```
""" This is the docstring at the beginning of a module
"""
from astropy import config as _config

class Conf(_config.ConfigNamespace):
    """
    Configuration parameters for my subpackage.
    """
    some_setting = _config.ConfigItem(
        1, 'Description of some_setting')
    another_setting = _config.ConfigItem(
        'string value', 'Description of another_setting')
# Create an instance for the user
conf = Conf()

... implementation ...
def some_func():
    #to get the value of these options, I might do:
    something = conf.some_setting + 2
    return conf.another_setting + ' Also, I added text.'
```

The configuration items also need to be added to the config file template. For `astropy`, this file is in `astropy/astropy.cfg`. For an affiliated package called, for example, `packagename`, the file is in `packagename/packagename.cfg`. For the example above, the following content would be added to the config file template:

```
[subpackage]
## Description of some_setting
# some_setting = 1

## Description of another_setting
# another_setting = foo
```

Note that the key/value pairs are commented out. This will allow for changing the default values in a future version of `astropy` without requiring the user to edit their configuration file to take advantage of the new defaults. By convention, the descriptions of each parameter are in comment lines starting with two hash characters (`##`) to distinguish them from commented out key/value pairs.

Item Types and Validation

If not otherwise specified, a **`ConfigItem`** gets its type from the type of the `defaultvalue` it is given when it is created. The item can only be set to be an object of this type. Hence:

```
some_setting = ConfigItem(1, 'A description.')
...
conf.some_setting = 1.2
```

will fail, because `1.2` is a float and `1` is an integer.

Note that if you want the configuration item to be limited to a particular set of options, you should pass in a list as the `defaultvalue` option. The first entry in the list will be taken as the default, and the list as a whole gives all of the valid options. For example:

```
an_option = ConfigItem(
    ['a', 'b', 'c'],
    "This option can be 'a', 'b', or 'c'")
...
conf.an_option = 'b'  # succeeds
conf.an_option = 'c'  # succeeds
conf.an_option = 'd'  # fails!
conf.an_option = 6    # fails!
```

Finally, a **`ConfigItem`** can be explicitly given a type via the `cfgtype` option:

```
an_int_setting = ConfigItem(
    1, 'A description.', cfgtype='integer')
...
conf.an_int_setting = 3     # works fine
conf.an_int_setting = 4.2   # fails!
```

If the default value's type does not match `cfgtype`, the **`ConfigItem`** cannot be created:

```
an_int_setting = ConfigItem(
    4.2, 'A description.', cfgtype='integer')
```

In summary, the default behavior (of automatically determining `cfgtype`) is usually what you want. The main exception is when you want your configuration item to be a list. The default behavior will treat that as a list of *options* unless you explicitly tell it that the **`ConfigItem`** itself is supposed to be a list:

```
a_list_setting = ConfigItem([1, 2, 3], 'A description.')

a_list_setting = ConfigItem([1, 2, 3], 'A description.',
cfgtype='list')
```

Details of all of the valid `cfgtype` items can be found in the validation section of the configobj manual. Below is a list of the valid values here for quick reference:

- 'integer'
- 'float'
- 'boolean'
- 'string'
- 'ip_addr'
- 'list'
- 'tuple'
- 'int_list'
- 'float_list'
- 'bool_list'
- 'string_list'
- 'ip_addr_list'
- 'mixed_list'
- 'option'
- 'pass'

Usage Tips

Keep in mind that `ConfigItem` objects can be changed at runtime by users. So it is always recommended to read their values immediately before use instead of storing their initial value to some other variable (or used as a default for a function). For example, the following will work, but is incorrect usage:

```
def some_func(val=conf.some_setting):
    return val + 2
```

This works fine as long as the user does not change its value during runtime, but if they do, the function will not know about the change:

```
>>> some_func()
3
>>> conf.some_setting = 3
>>> some_func()  # naively should return 5, because 3 + 2 = 5
3
```

There are two ways around this. The typical/intended way is:

```
def some_func():
    """
    The `SOME_SETTING` configuration item influences this output
    """
    return conf.some_setting + 2
```

Or, if the option needs to be available as a function parameter:

```
def some_func(val=None):
    """
    If not specified, `val` is set by the `SOME_SETTING`
configuration item.
    """
    return (conf.some_setting if val is None else val) + 2
```

Customizing Config in Affiliated Packages

The **astropy.config** package can be used by other packages. By default creating a config object in another package will lead to a configuration file taking the name of that package in the `astropy` config directory (i.e., `<astropy_config>/packagename.cfg`).

It is possible to configure this behavior so that the a custom configuration directory is created for your package, for example, `~/.packagename/packagename.cfg`. To do this, create a `packagename.config` subpackage and put the following into the `__init__.py` file:

```
import astropy.config as astropyconfig


class ConfigNamespace(astropyconfig.ConfigNamespace):
    rootname = 'packagename'


class ConfigItem(astropyconfig.ConfigItem):
    rootname = 'packagename'
```

Then replace all imports of **astropy.config** with `packagename.config`.

See Also

Astropy's Default Configuration File

```
## When True, use Unicode characters when outputting values,
and displaying
## widgets at the console.
# unicode_output = False

## When True, use ANSI color escape sequences when writing to
the console.
# use_color = True
```

```
## Maximum number of lines in the display of pretty-printed
objects. If not
## provided, try to determine automatically from the terminal
size.  Negative
## numbers mean no limit.
# max_lines = None

## Maximum number of characters per line in the display of
pretty-printed
## objects.  If not provided, try to determine automatically
from the terminal
## size. Negative numbers mean no limit.
# max_width = None

[io.fits]

## If True, enable support for record-valued keywords as
described by FITS WCS
## distortion paper. Otherwise they are treated as normal
keywords.
# enable_record_valued_keyword_cards = True

## If True, extension names (i.e. the ``EXTNAME`` keyword)
should be treated
## as case-sensitive.
# extension_name_case_sensitive = False

## If True, automatically remove trailing whitespace for
string values in
## headers.  Otherwise the values are returned verbatim, with
all whitespace
## intact.
# strip_header_whitespace = True

## If True, use memory-mapped file access to read/write the
data in FITS
## files. This generally provides better performance,
especially for large
## files, but may affect performance in I/O-heavy
applications.
# use_memmap = True

## If True, use lazy loading of HDUs when opening FITS files
by default; that
## is fits.open() will only seek for and read HDUs on demand
rather than
## reading all HDUs at once.  See the documentation for
fits.open() for more
## datails.
```

```
# lazy_load_hdus = True

## If True, default to recognizing the convention for
representing unsigned
## integers in FITS--if an array has BITPIX > 0, BSCALE = 1,
and BZERO =
## 2**BITPIX, represent the data as unsigned integers per
this convention.
# enable_uint = True

[io.votable]

## Can be 'exception' (treat fixable violations of the
VOTable spec as
## exceptions), 'warn' (show warnings for VOTable spec
violations), or
## 'ignore' (silently ignore VOTable spec violations)
# verify = ignore

[io.votable.exceptions]

## Number of times the same type of warning is displayed
before being
## suppressed
# max_warnings = 10

[logger]

## Threshold for the logging messages. Logging messages that
are less severe
## than this level will be ignored. The levels are
``'DEBUG'``, ``'INFO'``,
## ``'WARNING'``, ``'ERROR'``.
# log_level = INFO

## Whether to log `warnings.warn` calls.
# log_warnings = True

## Whether to log exceptions before raising them.
# log_exceptions = False

## Whether to always log messages to a log file.
# log_to_file = False

## The file to log messages to.  If empty string is given, it
defaults to a
## file ``'astropy.log'`` in the astropy config directory.
# log_file_path =
## Threshold for logging messages to `log_file_path`.
# log_file_level = INFO

## Format for log file entries.
# log_file_format = %(asctime)r, %(origin)r, %(levelname)r,
%(message)r

## The encoding (e.g., UTF-8) to use for the log file.  If
empty string is
## given, it defaults to the platform-preferred encoding.
# log_file_encoding =

[nddata]

## Whether to issue a warning if `~astropy.nddata.NDData`
arithmetic is
## performed with uncertainties and the uncertainties do not
support the
## propagation of correlated uncertainties.
# warn_unsupported_correlated = True

## Whether to issue a warning when the
`~astropy.nddata.NDData` unit attribute
## is changed from a non-``None`` value to another value that
data
## values/uncertainties are not scaled with the unit change.
# warn_setting_unit_directly = True

[samp]

## Whether to allow `astropy.samp` to use the internet, if
available.
# use_internet = True

## How many times to retry communications when they fail
# n_retries = 10

[table]

## The template that determines the name of a column if it
cannot be
## determined. Uses new-style (format method) string
formatting.
# auto_colname = col{0}

## The table class to be used in Jupyter notebooks when
displaying tables (and
## not overridden). See <https://getbootstrap.com/css/#tables
for a list of
```

```
## useful bootstrap classes.
# default_notebook_table_class = table-striped table-bordered
table-condensed

## List of conditions for issuing a warning when replacing a
table column
## using setitem, e.g. t['a'] = value.  Allowed options are
'always', 'slice',
## 'refcount', 'attributes'.
# replace_warnings = []

## Always use in-place update of a table column when using
setitem, e.g.
## t['a'] = value.  This overrides the default behavior of
replacing the
## column entirely with the new value when possible. This
configuration option
## will be deprecated and then removed in subsequent major
releases.
# replace_inplace = False

[table.jsviewer]

## The URL to the jquery library.
# jquery_url = https://code.jquery.com/jquery-3.1.1.min.js

## The URL to the jquery datatables library.
# datatables_url = https://cdn.datatables.net/1.10.12
/js/jquery.dataTables.min.js

## The URLs to the css file(s) to include.
# css_urls = ['https://cdn.datatables.net/1.10.12
/css/jquery.dataTables.css']

[time]

## Use fast C parser for supported time strings formats,
including ISO, ISOT,
## and YearDayTime. Allowed values are the 'False' (use
Python parser),'True'
## (use C parser and fall through to Python parser if fails),
and 'force' (use
## C parser and raise exception if it fails). Note that
theoptions are all
## strings.
# use_fast_parser = True

[units.quantity]
```

```
## The maximum size an array Quantity can be before its LaTeX
representation
## for IPython gets "summarized" (meaning only the first and
last few elements
## are shown with "..." between). Setting this to a negative
number means that
## the value will instead be whatever numpy gets from
get_printoptions.
# latex_array_threshold = 100

[utils.data]

## Primary URL for astropy remote data site.
# dataurl = http://data.astropy.org/

## Mirror URL for astropy remote data site.
# dataurl_mirror = http://www.astropy.org/astropy-data/

## Default User-Agent for HTTP request headers. This can be
overwritten for a
## particular call via http_headers option, where available.
This only
## provides the default value when not set by https_headers.
# default_http_user_agent = astropy

## Time to wait for remote data queries (in seconds). Set
this to zero to
## prevent any attempt to download anything (this will stop
working in a
## future release, use allow_internet=False instead).
# remote_timeout = 10.0

## If False, prevents any attempt to download from Internet.
# allow_internet = True

## Block size for computing file hashes.
# compute_hash_block_size = 65536

## Number of bytes of remote data to download per step.
# download_block_size = 65536

## Unused; cache no longer locked. Was: Number of seconds to
wait for the
## cache lock to be free. It should normally only ever be
held long enough to
## copy an already-downloaded file into the cache, so this
will normally only
## run over if something goes wrong and the lock is left held
by a dead
```

```
## process; the exception raised should indicate this and
what to do to fix
## it.
# download_cache_lock_attempts = 5

## If True, temporary download files created when the cache
is inaccessible
## will be deleted at the end of the python session.
# delete_temporary_downloads_at_exit = True

[utils.iers.iers]

## Enable auto-downloading of the latest IERS data.  If set
to False then the
## local IERS-B and leap-second files will be used by
default. Default is
## True.
# auto_download = True

## Maximum age (days) of predictive data before auto-
downloading. Default is
## 30.
# auto_max_age = 30.0

## URL for auto-downloading IERS file data.
# iers_auto_url =
ftp://anonymous:mail%40astropy.org@gdc.cddis.eosdis.nasa.gov
/pub/products/iers/finals2000A.all

## Mirror URL for auto-downloading IERS file data.
# iers_auto_url_mirror = https://datacenter.iers.org/data/9
/finals2000A.all

## Remote timeout downloading IERS file data (seconds).
# remote_timeout = 10.0

## System file with leap seconds.
# system_leap_second_file =

## URL for auto-downloading leap seconds.
# iers_leap_second_auto_url = https://hpiers.obspm.fr
/iers/bul/bulc/Leap_Second.dat

## Alternate URL for auto-downloading leap seconds.
# ietf_leap_second_auto_url = https://www.ietf.org/timezones
/data/leap-seconds.list

[visualization.wcsaxes]
```

```
## The number of samples along each image axis when
determining the range of
## coordinates in a plot.
# coordinate_range_samples = 50

## How many points to sample along the axes when determining
tick locations.
# frame_boundary_samples = 1000

## How many points to sample along grid lines.
# grid_samples = 1000

## The grid size to use when drawing a grid using contours
# contour_grid_samples = 200
```

Configuration Transition

This document describes the changes in the configuration system in `astropy` 0.4 and how to update code in order to use it.

For Users

The Config File

If you have not edited the configuration file in `~/.astropy/config/astropy.cfg`, there is nothing for you to do. The first time you import `astropy` 0.4, it will automatically be replaced with the configuration file template for `astropy` 0.4.

If you have edited the configuration file, it will be left untouched. However, the template for `astropy` 0.4 will be installed as `~/.astropy/config/astropy.0.4.cfg`. You can manually compare your changes to this file to determine what customizations should be brought over.

Saving

Saving configuration items from Python has been completely removed. Instead, the configuration file must be edited directly.

Renames

The location of the configuration parameters have been simplified, so they always appear in a high-level sub-package of `astropy`, rather than in low-level file names (which were really an implementation detail that should not have been exposed to the user). On the Python side, configuration items always are referenced through a `conf` object at the root of a sub-package.

Some configuration items that affect the results of science calculations have been removed as configuration parameters altogether and converted to science state objects that must be changed from Python code.

The following table lists all of the moves (in alphabetical order by original configuration file location). The old names will continue to work both from Python and the configuration file for the `astropy` 0.4 release cycle, and will be removed altogether in `astropy` 0.5.

Renamed configuration paramete

Old config file location	Old Python location	New conf
`[] unicode_output`	`UNICODE_OUTPUT`	
`[coordinates.name_resolve]` `name_resolve_timeout`	`coordinates.name_resolve.NAME_RESOLVE_TIMEOUT`	`[as` `r`
`[coordinates.name_resolve]` `sesame_url`	`coordinates.name_resolve.SESAME_URL`	
`[coordinates.name_resolve]` `sesame_database`	`coordinates.name_resolve.SESAME_DATABASE`	
`[cosmology.core]` `default_cosmology`	`cosmology.core.DEFAULT_COSMOLOGY`	
`[io.fits]` `enable_record_valued_keyword_cards`	`io.fits.ENABLE_RECORD_VALUED_KEYWORD_CARDS`	
`[io.fits]` `extension_name_case_sensitive`	`io.fits.EXTENSION_NAME_CASE_SENSITIVE`	
`[io.fits] strip_header_whitespace`	`io.fits.STRIP_HEADER_WHITESPACE`	
`[io.fits] use_memmap`	`io.fits.USE_MEMMAP`	
`[io.votable.table] pedantic`	`io.votable.table.PEDANTIC`	`[io.`
`[logger] log_exceptions`	`logger.LOG_EXCEPTIONS`	
`[logger] log_file_format`	`logger.LOG_FILE_FORMAT`	
`[logger] log_file_level`	`logger.LOG_FILE_LEVEL`	
`[logger] log_file_path`	`logger.LOG_FILE_PATH`	
`[logger] log_level`	`logger.LOG_LEVEL`	
`[logger] log_to_file`	`logger.LOG_TO_FILE`	
`[logger] log_warnings`	`logger.LOG_WARNINGS`	
`[logger] use_color`	`logger.USE_COLOR`	
`[nddata.nddata]` `warn_unsupported_correlated`	`nddata.nddata.WARN_UNSUPPORTED_CORRELATED`	`warn_un`

Old config file location	Old Python location	New conf
`[table.column] auto_colname`	`table.column.AUTO_COLNAME`	`[ta`
`[table.jsviewer] jquery_url`	`table.jsviewer.JQUERY_URL`	
`[table.jsviewer] datatables_url`	`table.jsviewer.DATATABLES_URL`	
`[table.pprint] max_lines`	`table.pprint.MAX_LINES`	
`[table.pprint] max_width`	`table.pprint.MAX_WIDTH`	
`[utils.console] use_color`	`utils.console.USE_COLOR`	
`[utils.data]` `compute_hash_block_size`	`astropy.utils.data.COMPUTE_HASH_BLOCK_SIZE`	
`[utils.data] dataurl`	`astropy.utils.data.DATAURL`	
`[utils.data]` `delete_temporary_downloads_at_exit`	`astropy.utils.data.DELETE_TEMPORARY_DOWNLOADS_AT_EXIT`	
`[utils.data]` `download_cache_block_size`	`astropy.utils.data.DOWNLOAD_CACHE_BLOCK_SIZE`	
`[utils.data]` `download_cache_lock_attempts`	`astropy.utils.data.download_cache_lock_attempts`	
`[utils.data] remote_timeout`	`astropy.utils.data.REMOTE_TIMEOUT`	
`[vo.client.conesearch]` `conesearch_dbname`	`vo.client.conesearch.CONESEARCH_DBNAME`	`[vo]`
`[vo.client.vos_catalog]` `vos_baseurl`	`vo.client.vos_catalog.BASEURL`	`[`
`[vo.samp.utils] use_internet`	`vo.samp.utils.ALLOW_INTERNET`	`[vo.`
`[vo.validator.validate]` `cs_mstr_list`	`vo.validator.validate.CS_MSTR_LIST`	`cones`
`[vo.validator.validate] cs_urls`	`vo.validator.validate.CS_URLS`	`c`
`[vo.validator.validate]` `noncrit_warnings`	`vo.validator.validate.noncrit_warnings`	`nonc`

For Affiliated Package Authors

For an affiliated package to support both `astropy` 0.3 and 0.4, following the `astropy` 0.3 config instructions should continue to work. Note that saving of configuration items has been removed entirely from `astropy` 0.4 without a deprecation cycle, so if saving configuration programmatically is important to your package, you may want to consider another method to save that state.

However, by the release of `astropy` 0.5, the `astropy` 0.3 config API will no longer work. The following describes how to transition an affiliated package written for `astropy` 0.3 to support `astropy` 0.4 and later. It will not be possible to support `astropy` 0.3, 0.4 and 0.5 simultaneously. Below `pkgname` is the name of your affiliated package.

The automatic generation of configuration files from the `ConfigurationItem` objects that it finds has been removed. Instead, the project should include a hard-coded “template” configuration file in `pkgname/pkgname.cfg`. By convention, and to ease upgrades for end users, all of the values should be commented out. For example:

```
[nddata]

## Whether to issue a warning if NDData arithmetic is performed with
## uncertainties and the uncertainties do not support the propagation
of
## correlated uncertainties.
# warn_unsupported_correlated = True
```

Affiliated packages should transition to using **`astropy.config.ConfigItem`** objects as members of **`astropy.config.ConfigNamespace`** subclasses.

For example, the following is an example of the `astropy` 0.3 and earlier method to define configuration items:

```
from astropy.config import ConfigurationItem

ENABLE_RECORD_VALUED_KEYWORD_CARDS = ConfigurationItem(
    'enabled_record_valued_keyword_cards', True,
    'If True, enable support for record-valued keywords as described
by '
    'the FITS WCS distortion paper. Otherwise they are treated as
normal '
    'keywords.')

EXTENSION_NAME_CASE_SENSITIVE = ConfigurationItem(
    'extension_name_case_sensitive', False,
    'If True, extension names (i.e. the EXTNAME keyword) should be '
   'treated as case-sensitive.')
```

The above, converted to the new method, looks like:

```
from astropy import config as _config

class Conf(_config.ConfigNamespace):
    """
    Configuration parameters for `astropy.io.fits`.
    """

    enable_record_valued_keyword_cards = _config.ConfigItem(
        True,
        'If True, enable support for record-valued keywords as
described by '
        'the FITS WCS distortion paper. Otherwise they are treated as
normal '
        'keywords.',
        aliases=
['astropy.io.fits.enabled_record_valued_keyword_cards'])

    extension_name_case_sensitive = _config.ConfigItem(
        False,
        'If True, extension names (i.e. the ``EXTNAME`` keyword)
should be '
        'treated as case-sensitive.')
conf = Conf()
```

Moving/Renaming Configuration Items in Python

`ConfigAlias` objects can be used when a configuration item has been moved from an `astropy` 0.3-style `ConfigurationItem` to an `astropy` 0.4-style `ConfigItem` inside of a `ConfigNamespace`.

In the above example, the following adds backward-compatible hooks so the old Python locations of the configuration items will continue to work from user code:

```
ENABLE_RECORD_VALUED_KEYWORD_CARDS = _config.ConfigAlias(
    '0.4', 'ENABLE_RECORD_VALUED_KEYWORD_CARDS',
    'enable_record_valued_keyword_cards')
```

Moving/Renaming Configuration Items in the Config File

If a configuration item is moved or renamed within the configuration file, the `aliases` kwarg to `ConfigItem` can be used so that the old location will continue to be used as a fallback. For example, if the old location of an item was:

```
[coordinates.name_resolve]
sesame_url = http://somewhere.com
```

You might want to drop the fact that this is implemented in the module `name_resolve` and just store the configuration in `coordinates`:

```
[coordinates]
sesame_url = http://somewhere.com
```

When defining the `ConfigItem` for this entry, the `aliases` kwarg can list the old location(s) of the configuration item:

```
sesame_url = _config.ConfigItem(
    ["http://somewhere.com"],
    """Docstring""",
    aliases=['astropy.coordinates.name_resolve.sesame_url'])
```

Logging system (overview of **`astropy.logger`**)

Reference/API

astropy.config Package

This module contains configuration and setup utilities for the Astropy project. This includes all functionality related to the affiliated package index.

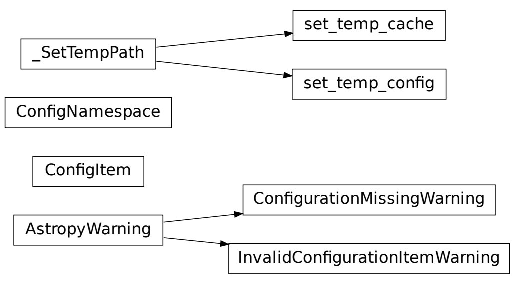
Functions

`generate_config`([pkgname, filename, verbose])	Generates a configuration file, from the list of **`ConfigItem`** objects for each subpackage.
`get_cache_dir`([rootname])	Determines the Astropy cache directory name and creates the directory if it doesn't exist.
`get_config`([packageormod, reload, rootname])	Gets the configuration object or section associated with a particular package or module.
`get_config_dir`([rootname])	Determines the package configuration directory name and creates the directory if it doesn't exist.
`reload_config`([packageormod, rootname])	Reloads configuration settings from a configuration file for the root package of the requested package/module.

Classes

`ConfigItem`([defaultvalue, description, …])	A setting and associated value stored in a configuration file.
`ConfigNamespace`()	A namespace of configuration items.
`ConfigurationMissingWarning`	A Warning that is issued when the configuration directory cannot be accessed (usually due to a permissions problem).
`InvalidConfigurationItemWarning`	A Warning that is issued when the configuration value specified in the astropy configuration file does not match the type expected for that configuration value.
`set_temp_cache`([path, delete])	Context manager to set a temporary path for the Astropy download cache, primarily for use with testing (though there may be other applications for setting a different cache directory, for example to switch to a cache dedicated to large files).
`set_temp_config`([path, delete])	Context manager to set a temporary path for the Astropy config, primarily for use with testing.

Class Inheritance Diagram



I/O Registry (`astropy.io.registry`)

Note

The I/O registry is only meant to be used directly by users who want to define their own custom readers/writers. Users who want to find out more about what built-in formats are supported by `Table` by default should see Unified File Read/Write Interface. No built-in formats are currently defined for `NDData`, but this will be added in future.

Introduction

The I/O registry is a submodule used to define the readers/writers available for the `Table` and `NDData` classes.

Using `astropy.io.registry`

This section demonstrates how to create a custom reader/writer. A reader is written as a function that can take any arguments except `format` (which is needed when manually specifying the format — see below) and returns an instance of the **Table** or **NDData** classes (or subclasses).

Examples

Here we assume that we are trying to write a reader/writer for the **Table** class:

```
from astropy.table import Table

def my_table_reader(filename, some_option=1):
    # Read in the table by any means necessary
    return table  # should be an instance of Table
```

Such a function can then be registered with the I/O registry:

```
from astropy.io import registry
registry.register_reader('my-table-format', Table, my_table_reader)
```

where the first argument is the name of the format, the second argument is the class that the function returns an instance for, and the third argument is the reader itself.

We can then read in a table with:

```
d = Table.read('my_table_file.mtf', format='my-table-format')
```

In practice, it would be nice to have the `read` method automatically identify that this file is in the `my-table-format` format, so we can construct a function that can recognize these files, which we refer to here as an *identifier* function.

An identifier function should take a first argument that is a string which indicates whether the identifier is being called from `read` or `write`, and should then accept an arbitrary number of positional and keyword arguments via `*args` and `**kwargs`, which are the arguments passed to the `read` method.

In the above case, we can write a function that only looks at filenames (but in practice, this function could even look at the first few bytes of the file, for example). The only requirement for the identifier function is that it return a boolean indicating whether the input matches that expected for the format. In our example, we want to automatically recognize files with filenames ending in `.mtf` as being in the `my-table-format` format:

```
import os
```

```
def identify_mtf(origin, *args, **kwargs):
    return (isinstance(args[0], str) and
            os.path.splitext(args[0].lower())[1] == '.mtf')
```

Note

Identifier functions should be prepared for arbitrary input — in particular, the first argument may not be a filename or file object, so it should not assume that this is the case.

We then register this identifier function, similarly to the reader function:

```
registry.register_identifier('my-table-format', Table, identify_mtf)
```

Having registered this function, we can then do:

```
t = Table.read('catalog.mtf')
```

If multiple formats match the current input, then an exception is raised, and similarly if no format matches the current input. In that case, the format should be explicitly given with the `format=` keyword argument.

It is also possible to create custom writers. To go with our custom reader above, we can write a custom writer:

```
def my_table_writer(table, filename, overwrite=False):
    ...  # Write the table out to a file
```

Writer functions should take a dataset object (either an instance of the **`Table`** or **`NDData`** classes or subclasses), and any number of subsequent positional and keyword arguments — although as for the reader, the `format` keyword argument cannot be used.

We then register the writer:

```
registry.register_writer('my-custom-format', Table, my_table_writer)
```

We can write the table out to a file:

```
t.write('catalog_new.mtf', format='my-table-format')
```

Since we have already registered the identifier function, we can also do:

```
t.write('catalog_new.mtf')
```

Reference/API

astropy.io.registry Module

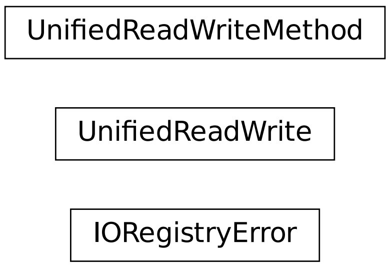
Functions

`register_reader`(data_format, data_class, …)	Register a reader function.
`register_writer`(data_format, data_class, …)	Register a table writer function.
`register_identifier`(data_format, data_class, …)	Associate an identifier function with a specific data type.
`identify_format`(origin, data_class_required, …)	Loop through identifiers to see which formats match.
`get_reader`(data_format, data_class)	Get reader for `data_format`.
`get_writer`(data_format, data_class)	Get writer for `data_format`.
`read`(cls, *args[, format, cache])	Read in data.
`write`(data, *args[, format])	Write out data.
`get_formats`([data_class, readwrite])	Get the list of registered I/O formats as a Table.
`delay_doc_updates`(cls)	Contextmanager to disable documentation updates when registering reader and writer.

Classes

`IORegistryError`	Custom error for registry clashes.
`UnifiedReadWriteMethod`(func)	Descriptor class for creating read() and write() methods in unified I/O.
`UnifiedReadWrite`(instance, cls, method_name)	Base class for the worker object used in unified read() or write() methods.

Class Inheritance Diagram



Logging system

Overview

The Astropy logging system is designed to give users flexibility in deciding which log messages to show, to capture them, and to send them to a file.

All messages printed by Astropy routines should use the built-in logging facility (normal `print()` calls should only be done by routines that are explicitly requested to print output). Messages can have one of several levels:

- DEBUG: Detailed information, typically of interest only when diagnosing problems.
- INFO: An message conveying information about the current task, and confirming that things are working as expected
- WARNING: An indication that something unexpected happened, and that user action may be required.
- ERROR: indicates a more serious issue, including exceptions

By default, only WARNING and ERROR messages are displayed, and are sent to a log file located at `~/.astropy/astropy.log` (if the file is writeable).

Configuring the logging system

First, import the logger:

```
from astropy import log
```

The threshold level (defined above) for messages can be set with e.g.:

```
log.setLevel('INFO')
```

Color (enabled by default) can be disabled with:

```
log.disable_color()
```

and enabled with:

```
log.enable_color()
```

Warnings from `warnings.warn` can be logged with:

```
log.enable_warnings_logging()
```

which can be disabled with:

```
log.disable_warnings_logging()
```

and exceptions can be included in the log with:

```
log.enable_exception_logging()
```

which can be disabled with:

```
log.disable_exception_logging()
```

It is also possible to set these settings from the Astropy configuration file, which also allows an overall log file to be specified. See Using the configuration file for more information.

Context managers

In some cases, you may want to capture the log messages, for example to check whether a specific message was output, or to log the messages from a specific section of code to a file. Both of these are possible using context managers.

To add the log messages to a list, first import the logger if you have not already done so:

```
from astropy import log
```

then enclose the code in which you want to log the messages to a list in a `with` statement:

```
with log.log_to_list() as log_list:
    # your code here
```

In the above example, once the block of code has executed, `log_list` will be a Python list containing all the Astropy logging messages that were raised. Note that messages continue to be output as normal.

Similarly, you can output the log messages of a specific section of code to a file using:

```
with log.log_to_file('myfile.log'):
    # your code here
```

which will add all the messages to `myfile.log` (this is in addition to the overall log file mentioned in Using the configuration file).

While these context managers will include all the messages emitted by the logger (using the global level set by `log.setLevel`), it is possible to filter a subset of these using `filter_level=`, and specifying one of `'DEBUG'`, `'INFO'`, `'WARN'`, `'ERROR'`. Note that if `filter_level` is a lower level than that set via `setLevel`, only messages with the level set by `setLevel` or higher will be included (i.e. `filter_level` is only filtering a subset of the messages normally emitted by the logger).

Similarly, it is possible to filter a subset of the messages by origin by specifying `filter_origin=` followed by a string. If the origin of a message starts with that string, the message will be included in the context manager. For example, `filter_origin='astropy.wcs'` will include only messages emitted in the `astropy.wcs` sub-package.

Using the configuration file

Options for the logger can be set in the `[logger]` section of the Astropy configuration file:

```
[logger]

# Threshold for the logging messages. Logging messages that are less
severe
# than this level will be ignored. The levels are 'DEBUG', 'INFO',
'WARNING',
# 'ERROR'
log_level = 'INFO'

# Whether to use color for the level names
use_color = True

# Whether to log warnings.warn calls
log_warnings = False

# Whether to log exceptions before raising them
log_exceptions = False

# Whether to always log messages to a log file
log_to_file = True

# The file to log messages to. If empty string is given, it defaults
to a
# file `astropy.log` in the astropy config directory.
log_file_path = '~/.astropy/astropy.log'

# Threshold for logging messages to log_file_path
log_file_level = 'INFO'
```

```
# Format for log file entries
log_file_format = '%(asctime)s, %(origin)s, %(levelname)s,
%(message)s'

# The encoding (e.g., UTF-8) to use for the log file.  If empty
string is
# given, it defaults to the platform-preferred encoding.
log_file_encoding = ""
```

Reference/API

astropy.logger Module

This module defines a logging class based on the built-in logging module

Classes

`Conf`()	Configuration parameters for `astropy.logger`.
`AstropyLogger`(name[, level])	This class is used to set up the Astropy logging.
`LoggingError`	This exception is for various errors that occur in the astropy logger, typically when activating or deactivating logger-related features.

Python warnings system

Astropy uses the Python `warnings` module to issue warning messages. The details of using the warnings module are general to Python, and apply to any Python software that uses this system. The user can suppress the warnings using the python command line argument `-W"ignore"` when starting an interactive python session. For example:

```
$ python -W"ignore"
```

The user may also use the command line argument when running a python script as follows:

```
$ python -W"ignore" myscript.py
```

It is also possible to suppress warnings from within a python script. For instance, the warnings issued from a single call to the `astropy.io.fits.writeto` function may be suppressed from within a Python script using the `warnings.filterwarnings` function as follows:

```
>>> import warnings
>>> from astropy.io import fits
>>> warnings.filterwarnings('ignore', category=UserWarning,
append=True)
>>> fits.writeto(filename, data, overwrite=True)
```

An equivalent way to insert an entry into the list of warning filter specifications for simple call `warnings.simplefilter`:

```
>>> warnings.simplefilter('ignore', UserWarning)
```

Astropy includes its own warning classes, `AstropyWarning` and `AstropyUserWarning`. All warnings from Astropy are based on these warning classes (see below for the distinction between them). One can thus ignore all warnings from Astropy (while still allowing through warnings from other libraries like Numpy) by using something like:

```
>>> from astropy.utils.exceptions import AstropyWarning
>>> warnings.simplefilter('ignore', category=AstropyWarning)
```

Warning filters may also be modified just within a certain context using the `warnings.catch_warnings` context manager:

```
>>> with warnings.catch_warnings():
...     warnings.simplefilter('ignore', AstropyWarning)
...     fits.writeto(filename, data, overwrite=True)
```

As mentioned above, there are actually *two* base classes for Astropy warnings. The main distinction is that `AstropyUserWarning` is for warnings that are *intended* for typical users (e.g. "Warning: Ambiguous unit", something that might be because of improper input). In contrast, `AstropyWarning` warnings that are *not* `AstropyUserWarning` may be for lower-level warnings more useful for developers writing code that *uses* Astropy (e.g., the deprecation warnings discussed below). So if you're a user that just wants to silence everything, the code above will suffice, but if you are a developer and want to hide development-related warnings from your users, you may wish to still allow through `AstropyUserWarning`.

Astropy also issues warnings when deprecated API features are used. If you wish to *squelch* deprecation warnings, you can start Python with `-Wi::Deprecation`. This sets all deprecation warnings to ignored. There is also an Astropy-specific `AstropyDeprecationWarning` which can be used to disable deprecation warnings from Astropy only.

See the CPython documentation for more information on the -W argument.

Astropy Core Package Utilities (`astropy.utils`)

Introduction

The `astropy.utils` package contains general-purpose utility functions and classes. Examples include data structures, tools for downloading and caching from URLs, and version intercompatibility functions.

This functionality is not astronomy-specific, but is intended primarily for use by Astropy developers. It is all safe for users to use, but the functions and classes are typically more complicated or specific to a particular need of Astropy.

Because of the mostly standalone and grab-bag nature of these utilities, they are generally best understood through their docstrings, and hence this documentation generally does not have detailed sections like the other packages. The exceptions are below:

IERS data access (`astropy.utils.iers`)

Introduction

The `iers` package provides access to the tables provided by the International Earth Rotation and Reference Systems (IERS) service, in particular files allowing interpolation of published UT1-UTC and polar motion values for given times. The UT1-UTC values are used in `astropy.time` to provide UT1 values, and the polar motions are used in `astropy.coordinates` to determine Earth orientation for celestial-to-terrestrial coordinate transformations.

Note

The package also provides machinery to track leap seconds. Since it generally should not be necessary to deal with those by hand, this is not discussed below. For details, see the documentation of `LeapSeconds`.

Getting started

Starting with astropy 1.2, the latest IERS values (which include approximately one year of predictive values) are automatically downloaded from the IERS service when required. This happens when a time or coordinate transformation needs a value which is not already available via the download cache. In most cases there is no need for invoking the **iers** classes oneself, but it is useful to understand the situations when a download will occur and how this can be controlled.

Basic usage

By default, the IERS data are managed via instances of the **IERS_Auto** class. These instances are created internally within the relevant time and coordinate objects during transformations. If the astropy data cache does not have the required IERS data file then astropy will request the file from the IERS service. This will occur the first time such a transform is done for a new setup or on a new machine. Here is an example that shows the typical download progress bar:

```
>>> from astropy.time import Time
>>> t = Time('2016:001')
>>> t.ut1
Downloading https://maia.usno.navy.mil/ser7/finals2000A.all
|==================================================================|
3.0M/3.0M (100.00%)         6s
<Time object: scale='ut1' format='yday' value=2016:001:00:00:00.082>
```

Note that you can forcibly clear the download cache as follows:

```
>>> from astropy.utils.data import clear_download_cache
>>> clear_download_cache()
```

The default IERS data used automatically is updated by the service every 7 days and includes transforms dating back to 1973-01-01.

Note

The **IERS_Auto** class contains machinery to ensure that the IERS table is kept up to date by auto-downloading the latest version as needed. This means that the IERS table is assured of having the state-of-the-art definitive and predictive values for Earth rotation. As a user it is **your responsibility** to understand the accuracy of IERS predictions if your science depends on that. If you request `UT1-UTC` or polar motions for times beyond the range of IERS table data then the nearest available values will be provided.

Configuration parameters

There are three configuration parameters that control the behavior of the automatic IERS downloading:

auto_download:

Enable auto-downloading of the latest IERS data. If set to `False` then the local IERS-B file will be used by default (even if the full IERS file with predictions was already downloaded and cached). This replicates the behavior prior to astropy 1.2. (default=True)

auto_max_age:

Maximum age of predictive data before auto-downloading (days). See next section for details. (default=30)

iers_auto_url:

URL for auto-downloading IERS file data

iers_auto_url_mirror:

Mirror URL for auto-downloading IERS file data.

remote_timeout:

Remote timeout downloading IERS file data (seconds)

Auto refresh behavior

The first time that one attempts a time or coordinate transformation that requires IERS data, the latest version of the IERS table (from 1973 through one year into the future) will be downloaded and stored in the astropy cache.

Transformations will then use the cached data file if possible. However, the `IERS_Auto` table is automatically updated in place from the network if the following two conditions a met when the table is queried for `UT1-UTC` or polar motion values:

- Any of the requested IERS values are *predictive*, meaning that they have been extrapolated into the future with a model that is fit to measured data. The IERS table contains approximately one year of predictive data from the time it is created.
- The first predictive values in the table are at least `conf.auto_max_age days` old relative to the current actual time (i.e. `Time.now()`). This means that the IERS table is out of date and a newer version can be found on the IERS service.

The IERS Service provides the default online table (set by `astropy.utils.iers.IERS_A_URL`) and updates the content once each 7 days. The default value of `auto_max_age` is 30 days to avoid unnecessary network access, but one can reduce this to as low as 10 days.

Working offline

If you are working without an internet connection and doing transformations that require IERS data, there are a couple of options.

Disable auto downloading

Here you can do:

```
>>> from astropy.utils import iers
>>> iers.conf.auto_download = False
```

In this case any transforms will use the bundled IERS-B data which covers the time range from 1962 to just before the astropy release date. Any transforms outside of this range will not be allowed.

Set the auto-download max age parameter

Only do this if you understand what you are doing, THIS CAN GIVE INACCURATE ANSWERS! Assuming you have previously been connected to the internet and have downloaded and cached the IERS auto values previously, then do the following:

```
>>> iers.conf.auto_max_age = None
```

This disables the check of whether the IERS values are sufficiently recent, and all the transformations (even those outside the time range of available IERS data) will succeed with at most warnings.

Direct table access

In most cases the automatic interface will suffice, but you may need to directly load and manipulate IERS tables. IERS-B values are provided as part of astropy and can be used to calculate time offsets and polar motion directly, or set up for internal use in further time and coordinate transformations. For example:

```
>>> from astropy.utils import iers
>>> t = Time('2010:001')
>>> iers_b = iers.IERS_B.open()
>>> iers_b.ut1_utc(t)
<Quantity 0.114033 s>
>>> iers.earth_orientation_table.set(iers_b)
<ScienceState earth_orientation_table: <IERS_B length=...>...>
>>> t.ut1.iso
'2010-01-01 00:00:00.114'
```

Instead of local copies of IERS files, one can also download them, using `iers.IERS_A_URL` (or `iers.IERS_A_URL_MIRROR`) and `iers.IERS_B_URL`, and then use those for future time and coordinate transformations (in this example, just for a single calculation, by using **`earth_orientation_table`** as a context manager):

```
>>> iers_a = iers.IERS_A.open(iers.IERS_A_URL)
>>> with iers.earth_orientation_table.set(iers_a):
...     print(t.ut1.iso)
2010-01-01 00:00:00.114
```

To reset to the default, pass in `None` (which is equivalent to passing in `iers.IERS_Auto.open()`):

```
>>> iers.earth_orientation_table.set(None)
<ScienceState earth_orientation_table: <IERS...>...>
```

To see the internal IERS data that gets used in astropy you can do the following:

```
>>> dat = iers.earth_orientation_table.get()
>>> type(dat)
<class 'astropy.utils.iers.iers.IERS...'>
>>> dat
<IERS_Auto length=16196>
 year month  day    MJD   PolPMFlag_A ... UT1Flag    PM_x      PM_y
PolPMFlag
                     d                ...          arcsec    arcsec
int64 int64 int64 float64     str1    ... unicode1 float64   float64
unicode1
----- ----- ----- ------- ----------- ... -------- -------- --------
---------
   73     1     2 41684.0           I ...        B    0.143    0.137
B
   73     1     3 41685.0           I ...        B    0.141    0.134
B
   73     1     4 41686.0           I ...        B    0.139    0.131
B
   73     1     5 41687.0           I ...        B    0.137    0.128
B
  ...   ...   ...     ...         ... ...      ...      ...      ...
...
   17     5     2 57875.0           P ...        P 0.007211  0.44884
P
   17     5     3 57876.0           P ...        P 0.008757 0.450321
P
   17     5     4 57877.0           P ...        P 0.010328 0.451777
P
   17     5     5 57878.0           P ...        P 0.011924 0.453209
P
   17     5     6 57879.0           P ...        P 0.013544 0.454617
P
```

The explanation for most of the columns can be found in the file named `iers.IERS_A_README`. The important columns of this table are MJD, UT1_UTC, UT1Flag, PM_x, PM_y, PolPMFlag:

```
>>> dat['MJD', 'UT1_UTC', 'UT1Flag', 'PM_x', 'PM_y', 'PolPMFlag']
<IERS_Auto length=16196>
  MJD     UT1_UTC    UT1Flag    PM_x     PM_y   PolPMFlag
   d         s                 arcsec   arcsec
float64  float64    unicode1 float64  float64    unicode1
------- ---------- -------- -------- -------- ---------
41684.0     0.8075        B    0.143    0.137         B
41685.0     0.8044        B    0.141    0.134         B
41686.0     0.8012        B    0.139    0.131         B
41687.0     0.7981        B    0.137    0.128         B
    ...        ...      ...      ...      ...       ...
57875.0 -0.6545408        P 0.007211  0.44884         P
57876.0 -0.6559528        P 0.008757 0.450321         P
57877.0 -0.6573705        P 0.010328 0.451777         P
57878.0 -0.6587712        P 0.011924 0.453209         P
57879.0  -0.660187        P 0.013544 0.454617         P
```

Downloadable Data Management (astropy.utils.data)

Introduction

A number of Astropy's tools work with data sets that are either awkwardly large (e.g., **solar_system_ephemeris**) or regularly updated (e.g., **IERS_B**) or both (e.g., **IERS_A**). This kind of data - authoritative data made available on the Web, and possibly updated from time to time - is reasonably common in astronomy. The Astropy Project therefore provides some tools for working with such data.

The primary tool for this is the `astropy` *cache*. This is a repository of downloaded data, indexed by the URL where it was obtained. The tool **download_file** and various other things built upon it can use this cache to request the contents of a URL, and (if they choose to use the cache) the data will only be downloaded if it is not already present in the cache. The tools can be instructed to obtain a new copy of data that is in the cache but has been updated online.

The `astropy` cache is stored in a centralized place (on Linux machines by default it is `$HOME/.astropy/cache`; see Configuration System (astropy.config) for more details). You can check its location on your machine:

```
>>> import astropy.config.paths
>>> astropy.config.paths.get_cache_dir()
'/home/burnell/.astropy/cache'
```

This centralization means that the cache is persistent and shared between all `astropy` runs in any virtualenv by one user on one machine (possibly more if your home directory is shared between multiple machines). This can dramatically accelerate `astropy` operations and reduce the load on servers, like those of the IERS, that were not designed for heavy Web traffic. If you find the cache has corrupted or outdated data in it, you can remove an entry or clear the whole thing with **`clear_download_cache`**.

The files in the cache directory are named according to a cryptographic hash of their URL (currently MD5, so in principle malevolent entities can cause collisions, though the security risks this poses are marginal at most). The modification times on these files normally indicate when they were last downloaded from the Internet.

Usage Within Astropy

For the most part, you can ignore the caching mechanism and rely on `astropy` to have the correct data when you need it. For example, precise time conversions and sky locations need measured tables of the Earth's rotation from the IERS. The table **`IERS_Auto`** provides the infrastructure for many of these calculations. It makes available Earth rotation parameters, and if you request them for a time more recent than its tables cover, it will download updated tables from the IERS. So for example asking what time it is in UT1 (a timescale that reflects the irregularity of the Earth's rotation) probably triggers a download of the IERS data:

```
>>> from astropy.time import Time
>>> Time.now().ut1
Downloading https://maia.usno.navy.mil/ser7/finals2000A.all
|============================================| 3.2M/3.2M (100.00%)
1s
<Time object: scale='ut1' format='datetime' value=2019-09-22
08:39:03.812731>
```

But running it a second time does not require any new download:

```
>>> Time.now().ut1
<Time object: scale='ut1' format='datetime' value=2019-09-22
```

```
08:41:21.588836>
```

Some data is also made available from the Astropy data server either for use within `astropy` or for your convenience. These are available more conveniently with the `get_pkg_data_*` functions:

```
>>> from astropy.utils.data import get_pkg_data_contents
>>> print(get_pkg_data_contents("coordinates/sites-un-ascii"))
# these are all mappings from the name in sites.json (which is ASCII-
only) to the "true" unicode names
TUBITAK->TÜBİTAK
```

Usage From Outside Astropy

Users of `astropy` can also make use of `astropy`'s caching and downloading mechanism. In its simplest form, this amounts to using **download_file** with the `cache=True` argument to obtain their data, from the cache if the data is there:

```
>>> from astropy.utils.iers import IERS_B_URL, IERS_B
>>> from astropy.utils.data import download_file
>>> IERS_B.open(download_file(IERS_B_URL, cache=True))
["year","month","day"][-3:]
 <IERS_B length=3>
 year month  day
int64 int64 int64
----- ----- -----
 2019     8     4
 2019     8     5
 2019     8     6
```

If users want to update the cache to a newer version of the data (note that here the data was already up to date; users will have to decide for themselves when to obtain new versions), they can use the `cache='update'` argument:

```
>>> IERS_B.open(download_file(IERS_B_URL,
...                           cache='update')
... )["year","month","day"][-3:]
Downloading http://hpiers.obspm.fr/iers/eop/eopc04/eopc04_IAU2000.62-
now
|=========================================| 3.2M/3.2M (100.00%)
0s
<IERS_B length=3>
```

```
 year month  day 
int64 int64 int64
----- ----- -----
 2019     8    18
 2019     8    19
 2019     8    20
```

If they are concerned that the primary source of the data may be overloaded or unavailable, they can use the `sources` argument to provide a list of sources to attempt downloading from, in order. This need not include the original source. Regardless, the data will be stored in the cache under the original URL requested:

```
>>> f = download_file("ftp://ssd.jpl.nasa.gov/pub/eph/planets
/bsp/de405.bsp",
...     cache=True,
...     sources=['https://data.nanograv.org/static/data/ephem
/de405.bsp',
...              'ftp://ssd.jpl.nasa.gov/pub/eph/planets
/bsp/de405.bsp'])
Downloading ftp://ssd.jpl.nasa.gov/pub/eph/planets/bsp/de405.bsp from
https://data.nanograv.org/static/data/ephem/de405.bsp
|========================================|  65M/ 65M (100.00%)
19s
```

Cache Management

Because the cache is persistent, it is possible for it to become inconveniently large, or become filled with irrelevant data. While it is simply a directory on disk, each file is supposed to represent the contents of a URL, and many URLs do not make acceptable on-disk filenames (for example, containing troublesome characters like “:” and “~”). There is reason to worry that multiple `astropy` processes accessing the cache simultaneously might lead to cache corruption. The data is therefore stored in a subdirectory named after the hash of the URL, and write access is handled in a way that is resistant to concurrency problems. So access to the cache is more convenient with a few helpers provided by **`data`**.

If your cache starts behaving oddly you can use **`check_download_cache`** to examine your cache contents and raise an exception if it finds any anomalies. If a single file is undesired or damaged, it can be removed by calling **`clear_download_cache`** with an argument that is the URL it was obtained from, the filename of the downloaded file, or the hash of its contents. Should the cache ever become badly corrupted, `clear_download_cache` with no arguments will simply delete the whole directory, freeing the space and removing any inconsistent data. Of course, if you remove data using either of these tools, any processes currently using that data may be disrupted (or, under Windows, deleting the cache may not be possible until those processes terminate). So use `clear_download_cache` with care.

To check the total space occupied by the cache, use `cache_total_size`. The contents of the cache can be listed with `get_cached_urls`, and the presence of a particular URL in the cache can be tested with `is_url_in_cache`. More general manipulations can be carried out using `cache_contents`, which returns a `dict` mapping URLs to on-disk filenames of their contents.

If you want to transfer the cache to another computer, or preserve its contents for later use, you can use the functions `export_download_cache` to produce a ZIP file listing some or all of the cache contents, and `import_download_cache` to load the `astropy` cache from such a ZIP file.

The Astropy cache has changed format - once in the Python 2 to Python 3 transition, and again before Astropy version 4.0.2 to resolve some concurrency problems that arose on some compute clusters. Each version of the cache is in its own subdirectory, so the old versions do not interfere with the new versions and vice versa, but their contents are not used by this version and are not cleared by `clear_download_cache`. To remove these old cache directories, you can run:

```
>>> from shutil import rmtree
>>> from os.path import join
>>> from astropy.config.paths import get_cache_dir
>>> rmtree(join(get_cache_dir(), 'download', 'py2'),
ignore_errors=True)
>>> rmtree(join(get_cache_dir(), 'download', 'py3'),
ignore_errors=True)
```

Using Astropy With Limited or No Internet Access

You might want to use `astropy` on a telescope control machine behind a strict firewall. Or you might be running continuous integration (CI) on your `astropy` server and want to avoid hammering astronomy servers on every pull request for every architecture. Or you might not have access to US government or military web servers. Whichever is the case, you may need to avoid `astropy` needing data from the Internet. There is no simple and complete solution to this problem at the moment, but there are tools that can help.

Exactly which external data your project depends on will depend on what parts of `astropy` you use and how. The most general solution is to use a computer that can access the Internet to run a version of your calculation that pulls in all of the data files you will require, including sufficiently up-to-date versions of files like the IERS data that update regularly. Then once the cache on this connected machine is loaded with everything necessary, transport the cache contents to your target machine by whatever means you have available, whether by copying via an intermediate machine, portable disk drive, or some other tool. The cache directory itself is somewhat portable between machines of the same UNIX flavour; this may be sufficient if you can persuade your CI system to cache the directory between runs. For greater portability, though, you can simply use **`export_download_cache`** and **`import_download_cache`**, which are portable and will allow adding files to an existing cache directory.

If your application needs IERS data specifically, you can download the appropriate IERS table, covering the appropriate time span, by any means you find convenient. You can then load this file into your application and use the resulting table rather than **`IERS_Auto`**. In fact, the IERS B table is small enough that a version (not necessarily recent) is bundled with `astropy` as `astropy.utils.iers.IERS_B_FILE`. Using a specific non-automatic table also has the advantage of giving you control over exactly which version of the IERS data your application is using. See also Working offline.

If your issue is with certain specific servers, even if they are the ones `astropy` normally uses, if you can anticipate exactly which files will be needed (or just pick up after `astropy` fails to obtain them) and make those files available somewhere else, you can request they be downloaded to the cache using **`download_file`** with the `sources` argument set to locations you know do work. You can also set `sources` to an empty list to ensure that **`download_file`** does not attempt to use the Internet at all.

If you have a particular URL that is giving you trouble, you can download it using some other tool (e.g., `wget`), possibly on another machine, and then use **`import_file_to_cache`**.

Astropy Data and Clusters

Astronomical calculations often require the use of a large number of different processes on different machines with a shared home filesystem. This can pose certain complexities. In particular, if the many different processes attempt to download a file simultaneously this can overload a server or trigger security systems. The parallel access to the home directory can also trigger concurrency problems in the Astropy data cache, though we have tried to minimize these. We therefore recommend the following guidelines:

- Write a simple script that sets `astropy.utils.iers.conf.auto_download = True` and then accesses all cached resources your code will need, including source name lookups and IERS tables. Run it on the head node from time to time (frequently enough to beat the timeout `astropy.utils.iers.conf.auto_max_age`, which defaults to 30 days) to ensure all data is up to date.
- Make an Astropy config file (see Configuration System (astropy.config)) that sets `astropy.utils.iers.conf.auto_download = False` so that the worker jobs will not suddenly notice an out-of-date table all at once and frantically attempt to download it.
- Optionally, in this file, set `astropy.utils.data.conf.allow_internet = False` to prevent any attempt to download any file from the worker nodes; if you do this, you will need to override this setting in your script that does the actual downloading.

Now your worker nodes should not need to obtain anything from the Internet and all should run smoothly.

Note

The `astropy.utils.compat` subpackage is not included in this documentation. It contains utility modules for compatibility with older/newer versions of python and numpy, as well as including some bugfixes for the stdlib that are important for `astropy`. It is recommended that developers at least glance over the source code for this subpackage, but most of it cannot be reliably included here because of the large amount of version-specific code it contains. Its content is solely for internal use of `astropy` and subject to changes without deprecations. Do not use it in external packages or code.

Reference/API

astropy.utils.codegen Module

Utilities for generating new Python code at runtime.

Functions

make_function_with_signature(func[, args, …])	Make a new function from an existing function but with the desired signature.

astropy.utils.collections Module

A module containing specialized collection classes.

Classes

HomogeneousList(types[, values])	A subclass of list that contains only elements of a given type or types.

astropy.utils.console Module

Utilities for console input and output.

Functions

isatty(file)	Returns True if `file` is a tty.
color_print(*args[, end])	Prints colors and styles to the terminal uses ANSI escape sequences.
human_time(seconds)	Returns a human-friendly time string that is always exactly 6 characters long.
human_file_size(size)	Returns a human-friendly string representing a file size that is 2-4 characters long.
print_code_line(line[, col, file, tabwidth, …])	Prints a line of source code, highlighting a particular character position in the line.
terminal_size([file])	Returns a tuple (height, width) containing the height and width of the terminal.

Classes

ProgressBar(total_or_items[, …])	A class to display a progress bar in the terminal.
Spinner(msg[, color, file, step, chars])	A class to display a spinner in the terminal.
ProgressBarOrSpinner(total, msg[, color, file])	A class that displays either a ProgressBar or Spinner depending on whether the total size of the operation is known or not.

astropy.utils.data_info Module

This module contains functions and methods that relate to the DataInfo class which provides a container for informational attributes as well as summary info methods.

A DataInfo object is attached to the Quantity, SkyCoord, and Time classes in astropy. Here it allows those classes to be used in Tables and uniformly carry table column attributes such as name, format, dtype, meta, and description.

Functions

`data_info_factory`(names, funcs)	Factory to create a function that can be used as an `option` for outputting data object summary information.
`dtype_info_name`(dtype)	Return a human-oriented string name of the `dtype` arg.

Classes

`BaseColumnInfo`([bound])	Base info class for anything that can be a column in an astropy Table.
`DataInfo`([bound])	Descriptor that data classes use to add an `info` attribute for storing data attributes in a uniform and portable way.
`MixinInfo`([bound])	
`ParentDtypeInfo`([bound])	Mixin that gets info.dtype from parent

astropy.utils.decorators Module

Sundry function and class decorators.

Functions

`deprecated`(since[, message, name, …])	Used to mark a function or class as deprecated.
`deprecated_attribute`(name, since[, message, …])	Used to mark a public attribute as deprecated.
`deprecated_renamed_argument`(old_name, …[, …])	Deprecate a _renamed_ or _removed_ function argument.
`format_doc`(docstring, *args, **kwargs)	Replaces the docstring of the decorated object and then formats it.

Classes

classproperty([fget, doc, lazy])	Similar to **property**, but allows class-level properties.
lazyproperty(fget[, fset, fdel, doc])	Works similarly to property(), but computes the value only once.
sharedmethod	This is a method decorator that allows both an instancemethod and a **classmethod** to share the same name.

astropy.utils.diff Module

Functions

diff_values(a, b[, rtol, atol])	Diff two scalar values.
report_diff_values(a, b[, fileobj, indent_width])	Write a diff report between two values to the specified file-like object.
where_not_allclose(a, b[, rtol, atol])	A version of **numpy.allclose()** that returns the indices where the two arrays differ, instead of just a boolean value.

astropy.utils.exceptions Module

This module contains errors/exceptions and warnings of general use for astropy. Exceptions that are specific to a given subpackage should *not* be here, but rather in the particular subpackage.

Classes

AstropyWarning	The base warning class from which all Astropy warnings should inherit.
AstropyUserWarning	The primary warning class for Astropy.
AstropyDeprecationWarning	A warning class to indicate a deprecated feature.
AstropyPendingDeprecationWarning	A warning class to indicate a soon-to-be deprecated feature.
AstropyBackwardsIncompatibleChangeWarning	A warning class indicating a change in astropy that is incompatible with previous versions.
DuplicateRepresentationWarning	A warning class indicating a represenation name was already registered

astropy.utils.iers Package

Classes

`Conf`()	Configuration parameters for `astropy.utils.iers`.
`IERS`([data, masked, names, dtype, meta, …])	Generic IERS table class, defining interpolation functions.
`IERSRangeError`	Any error for when dates are outside of the valid range for IERS
`IERSStaleWarning`	
`IERS_A`([data, masked, names, dtype, meta, …])	IERS Table class targeted to IERS A, provided by USNO.
`IERS_Auto`([data, masked, names, dtype, …])	Provide most-recent IERS data and automatically handle downloading of updated values as necessary.
`IERS_B`([data, masked, names, dtype, meta, …])	IERS Table class targeted to IERS B, provided by IERS itself.
`LeapSeconds`([data, masked, names, dtype, …])	Leap seconds class, holding TAI-UTC differences.
`earth_orientation_table`()	Default IERS table for Earth rotation and reference systems service.

astropy.utils.introspection Module

Functions related to Python runtime introspection.

Functions

`resolve_name`(name, *additional_parts)	Resolve a name like `module.object` to an object and return it.
`minversion`(module, version[, inclusive, …])	Returns `True` if the specified Python module satisfies a minimum version requirement, and `False` if not.
`find_current_module`([depth, finddiff])	Determines the module/package from which this function is called.
`isinstancemethod`(cls, obj)	Returns `True` if the given object is an instance method of the class it is defined on (as opposed to a `staticmethod` or a `classmethod`).

astropy.utils.metadata Module

This module contains helper functions and classes for handling metadata.

Functions

`common_dtype`(arrs)	Use numpy to find the common dtype for a list of ndarrays.
`enable_merge_strategies`(*merge_strategies)	Context manager to temporarily enable one or more custom metadata merge strategies.

merge(left, right[, merge_func, …])	Merge the `left` and `right` metadata objects.

Classes

MergeConflictError	
MergeConflictWarning	
MergePlus()	Merge `left` and `right` objects using the plus operator.
MergeNpConcatenate()	Merge `left` and `right` objects using np.concatenate.
MergeStrategy()	Base class for defining a strategy for merging metadata from two sources, left and right, into a single output.
MergeStrategyMeta(name, bases, members)	Metaclass that registers MergeStrategy subclasses into the MERGE_STRATEGIES registry.
MetaData([doc, copy])	A descriptor for classes that have a `meta` property.
MetaAttribute([default])	Descriptor to define custom attribute which gets stored in the object `meta` dict and can have a defined default.

astropy.utils.misc Module

A “grab bag” of relatively small general-purpose utilities that don’t have a clear module/package to live in.

Functions

isiterable(obj)	Returns **True** if the given object is iterable.
silence()	A context manager that silences sys.stdout and sys.stderr.
format_exception(msg, *args, **kwargs)	Given an exception message string, uses new-style formatting arguments `{filename}`, `{lineno}`, `{func}` and/or `{text}` to fill in information about the exception that occurred.
find_api_page(obj[, version, openinbrowser, …])	Determines the URL of the API page for the specified object, and optionally open that page in a web browser.
is_path_hidden(filepath)	Determines if a given file or directory is hidden.
walk_skip_hidden(top[, onerror, followlinks])	A wrapper for **os.walk** that skips hidden files and directories.
indent(s[, shift, width])	Indent a block of text.
dtype_bytes_or_chars(dtype)	Parse the number out of a dtype.str value like ‘<U5’ or ‘<f8’.

Classes

`NumpyRNGContext`(seed)	A context manager (for use with the `with` statement) that will seed the numpy random number generator (RNG) to a specific value, and then restore the RNG state back to whatever it was before.
`JsonCustomEncoder`(*[, skipkeys, …])	Support for data types that JSON default encoder does not do.
`OrderedDescriptor`(*args, **kwargs)	Base class for descriptors whose order in the class body should be preserved.
`OrderedDescriptorContainer`(cls_name, bases, …)	Classes should use this metaclass if they wish to use `OrderedDescriptor` attributes, which are class attributes that “remember” the order in which they were defined in the class body.

astropy.utils.state Module

A simple class to manage a piece of global science state. See Adding New Configuration Items for more details.

Classes

`ScienceState`()	Science state subclasses are used to manage global items that can affect science results.

astropy.utils.shapes Module

The ShapedLikeNDArray mixin class and shape-related functions.

Functions

`check_broadcast`(*shapes)	Determines whether two or more Numpy arrays can be broadcast with each other based on their shape tuple alone.
`unbroadcast`(array)	Given an array, return a new array that is the smallest subset of the original array that can be re-broadcasted back to the original array.

Classes

`ShapedLikeNDArray`()	Mixin class to provide shape-changing methods.
`IncompatibleShapeError`(shape_a, shape_a_idx, …)	

File Downloads

astropy.utils.data Module

Functions for accessing, downloading, and caching data files.

Functions

`download_file`(remote_url[, cache, …])	Downloads a URL and optionally caches the result.
`download_files_in_parallel`(urls[, cache, …])	Download multiple files in parallel from the given URLs.
`get_readable_fileobj`(name_or_obj[, …])	Yield a readable, seekable file-like object from a file or URL.
`get_pkg_data_fileobj`(data_name[, package, …])	Retrieves a data file from the standard locations for the package and provides the file as a file-like object that reads bytes.
`get_pkg_data_filename`(data_name[, package, …])	Retrieves a data file from the standard locations for the package and provides a local filename for the data.
`get_pkg_data_contents`(data_name[, package, …])	Retrieves a data file from the standard locations and returns its contents as a bytes object.
`get_pkg_data_fileobjs`(datadir[, package, …])	Returns readable file objects for all of the data files in a given directory that match a given glob pattern.
`get_pkg_data_filenames`(datadir[, package, …])	Returns the path of all of the data files in a given directory that match a given glob pattern.
`is_url`(string)	Test whether a string is a valid URL for `download_file()`.
`is_url_in_cache`(url_key[, pkgname])	Check if a download for `url_key` is in the cache.
`get_cached_urls`([pkgname])	Get the list of URLs in the cache.
`cache_total_size`([pkgname])	Return the total size in bytes of all files in the cache.
`cache_contents`([pkgname])	Obtain a dict mapping cached URLs to filenames.
`export_download_cache`(filename_or_obj[, …])	Exports the cache contents as a ZIP file.
`import_download_cache`(filename_or_obj[, …])	Imports the contents of a ZIP file into the cache.
`import_file_to_cache`(url_key, filename[, …])	Import the on-disk file specified by filename to the cache.
`check_download_cache`([check_hashes, pkgname])	Do a consistency check on the cache.
`clear_download_cache`([hashorurl, pkgname])	Clears the data file cache by deleting the local file(s).
`compute_hash`(localfn)	Computes the MD5 hash for a file.
`get_free_space_in_dir`(path[, unit])	Given a path to a directory, returns the amount of free space on that filesystem.

`check_free_space_in_dir`(path, size)	Determines if a given directory has enough space to hold a file of a given size.
`get_file_contents`(*args, **kwargs)	Retrieves the contents of a filename or file-like object.

Classes

`Conf`()	Configuration parameters for `astropy.utils.data`.
`CacheMissingWarning`	This warning indicates the standard cache directory is not accessible, with the first argument providing the warning message.

XML

The `astropy.utils.xml.*` modules provide various XML processing tools.

astropy.utils.xml.check Module

A collection of functions for checking various XML-related strings for standards compliance.

Functions

`check_anyuri`(uri)	Returns `True` if *uri* is a valid URI as defined in RFC 2396.
`check_id`(ID)	Returns `True` if *ID* is a valid XML ID.
`check_mime_content_type`(content_type)	Returns `True` if *content_type* is a valid MIME content type (syntactically at least), as defined by RFC 2045.
`check_token`(token)	Returns `True` if *token* is a valid XML token, as defined by XML Schema Part 2.
`fix_id`(ID)	Given an arbitrary string, create one that can be used as an xml id.

astropy.utils.xml.iterparser Module

This module includes a fast iterator-based XML parser.

Functions

`get_xml_iterator`(source[, …])	Returns an iterator over the elements of an XML file.

`get_xml_encoding`(source)	Determine the encoding of an XML file by reading its header.
`xml_readlines`(source)	Get the lines from a given XML file.

astropy.utils.xml.unescaper Module

URL unescaper functions.

Functions

`unescape_all`(url)	Recursively unescape a given URL.

astropy.utils.xml.validate Module

Functions to do XML schema and DTD validation. At the moment, this makes a subprocess call to xmllint. This could use a Python-based library at some point in the future, if something appropriate could be found.

Functions

`validate_schema`(filename, schema_file)	Validates an XML file against a schema or DTD.

astropy.utils.xml.writer Module

Contains a class that makes it simple to stream out well-formed and nicely-indented XML.

Classes

`XMLWriter`(file)	A class to write well-formed and nicely indented XML.

Astropy Testing Tools

This section is primarily a reference for developers that want to understand or add to the Astropy testing machinery. See Testing Guidelines for an overview of running or writing the tests.

`astropy.tests.helper` Module

To ease development of tests that work with Astropy, the **astropy.tests.helper** module provides some utility functions to make tests that use Astropy conventions or classes easier to work with, e.g., functions to test for near-equality of **Quantity** objects.

The functionality here is not exhaustive, because much of the useful tools are either in the standard library, pytest, or numpy.testing. This module contains primarily functionality specific to the astropy core package or packages that follow the Astropy package template.

Reference/API

astropy.tests.helper Module

Functions

`enable_deprecations_as_exceptions`([…])	Turn on the feature that turns deprecations into exceptions.
`treat_deprecations_as_exceptions`()	Turn all DeprecationWarnings (which indicate deprecated uses of Python itself or Numpy, but not within Astropy, where we use our own deprecation warning class) into exceptions so that we find out about them early.
`assert_follows_unicode_guidelines`(x[, roundtrip])	Test that an object follows our Unicode policy.
`assert_quantity_allclose`(actual, desired[, …])	Raise an assertion if two objects are not equal up to desired tolerance.
`check_pickling_recovery`(original, protocol)	Try to pickle an object.
`pickle_protocol`(request)	Fixture to run all the tests for protocols 0 and 1, and -1 (most advanced).
`generic_recursive_equality_test`(a, b, …)	Check if the attributes of a and b are equal.

Classes

`raises`(exc)	A decorator to mark that a test should raise a given exception. Use as follows::.
`catch_warnings`(*classes)	A high-powered version of warnings.catch_warnings to use for testing and to make sure that there is no dependence on the order in which the tests are run.

Astropy Test Runner

When executing tests with `astropy.test` the call to pytest is controlled by the `astropy.tests.runner.TestRunner` class.

The `TestRunner` class is used to generate the `astropy.test` function, the test function generates a set of command line arguments to pytest. The arguments to pytest are defined in the `run_tests` method, the arguments to `run_tests` and their respective logic are defined in methods of `TestRunner` decorated with the `keyword` decorator. For an example of this see `TestRunnerBase`. This design makes it easy for packages to add or remove keyword arguments to their test runners, or define a whole new set of arguments by subclassing from `TestRunnerBase`.

Reference/API

class `astropy.tests.runner.`**`keyword`**(*default_value=None*, *priority=0*)

A decorator to mark a method as keyword argument for the `TestRunner`.

Parameters: **default_value** : *`object`*

The default value for the keyword argument. (Default: `None`)

priority : *`int`*

keyword argument methods are executed in order of descending priority.

class `astropy.tests.runner.`**`TestRunnerBase`**(**args*, ***kwargs*)

The base class for the TestRunner.

A test runner can be constructed by creating a subclass of this class and defining 'keyword' methods. These are methods that have the `keyword` decorator, these methods are used to construct allowed keyword arguments to the `run_tests` method as a way to allow customization of individual keyword arguments (and associated logic) without having to re-implement the whole `run_tests` method.

Examples

A simple keyword method:

```
class MyRunner(TestRunnerBase):

    @keyword('default_value'):
    def spam(self, spam, kwargs):
        """
        spam : `str`
            The parameter description for the run_tests docstring.
        """
        # Return value must be a list with a CLI parameter for
pytest.
        return ['--spam={}'.format(spam)]
```

class `astropy.tests.runner.` **`TestRunner`** (**args*, ***kwargs*)

A test runner for astropy tests

Try the development version

Note

git is the name of a source code management system. It is used to keep track of changes made to code and to manage contributions coming from several different people. If you want to read more about git right now take a look at Git Basics.

If you have never used git before, allow one hour the first time you do this. If you find this taking more than one hour, post in one of the astropy forums to get help.

Trying out the development version of astropy is useful in three ways:

- More users testing new features helps uncover bugs before the feature is released.
- A bug in the most recent stable release might have been fixed in the development version. Knowing whether that is the case can make your bug reports more useful.
- You will need to go through all of these steps before contributing any code to Astropy. Practicing now will save you time later if you plan to contribute.

Overview

Conceptually, there are several steps to getting a working copy of the latest version of astropy on your computer:

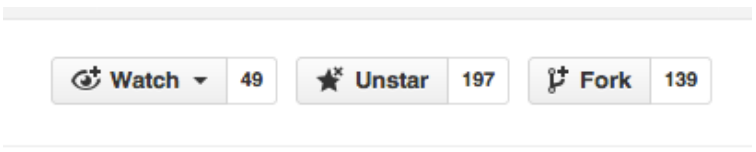
1. Make your own copy of Astropy on GitHub; this copy is called a *fork* (if you don’t have an account on github yet, go there now and make one).
2. Make sure git is installed and configured on your computer
3. Copy your fork of Astropy from GitHub to your computer; this is called making a *clone* of the repository.
4. Tell git where to look for changes in the development version of Astropy
5. Create your own private workspace; this is called making a *branch*.
6. “Activate” the development version of astropy
7. Test your development copy
8. Try out the development version
9. “Deactivate” the development version

Step-by-step instructions

Make your own copy of Astropy on GitHub

In the language of GitHub, making a copy of someone's code is called making a *fork*. A fork is a complete copy of the code and all of its revision history.

1. Log into your GitHub account.
2. Go to the Astropy GitHub home page.
3. Click on the *fork* button:



After a short pause and an animation of Octocat scanning a book on a flatbed scanner, you should find yourself at the home page for your own forked copy of astropy.

Make sure git is installed and configured on your computer

Check that git is installed:

Check by typing, in a terminal:

```
$ git --version
# if git is installed, will get something like: git version 2.20.1
```

If git is not installed, get it.

Basic git configuration:

Follow the instructions at Set Up Git at GitHub to take care of two essential items:

- Set your user name and email in your copy of git
- Set up authentication so you don't have to type your github password every time you need to access github from the command line. The default method at Set Up Git at GitHub may require administrative privileges; if that is a problem, set up authentication using SSH keys instead

We also recommend setting up git so that when you copy changes from your computer to GitHub only the copy (called a *branch*) of astropy that you are working on gets pushed up to GitHub. *If* your version of git is 1.7.11 or, greater, you can do that with:

```
git config --global push.default simple
```

If you skip this step now it is not a problem; git will remind you to do it in those cases when it is relevant. If your version of git is less than 1.7.11, you can still continue without this, but it may lead to confusion later, as you might push up branches you do not intend to push.

Note

Make sure you make a note of which authentication method you set up because it affects the command you use to copy your GitHub fork to your computer.

If you set up password caching (the default method) the URLs will look like `https://github.com/your-user-name/astropy.git`.

If you set up SSH keys the URLs you use for making copies will look something like `git@github.com:your-user-name/astropy.git`.

Copy your fork of Astropy from GitHub to your computer

One of the commands below will make a complete copy of your GitHub fork of Astropy in a directory called `astropy`; which form you use depends on what kind of authentication you set up in the previous step:

```
# Use this form if you setup SSH keys...
$ git clone --recursive git@github.com:your-user-name/astropy.git
# ...otherwise use this form:
$ git clone --recursive https://github.com/your-user-name/astropy.git
```

If there is an error at this stage it is probably an error in setting up authentication.

Tell git where to look for changes in the development version of Astropy

Right now your local copy of astropy doesn't know where the development version of astropy is. There is no easy way to keep your local copy up to date. In git the name for another location of the same repository is a *remote*. The repository that contains the latest "official" development version is traditionally called the *upstream* remote, but here we use a more meaningful name for the remote: *astropy*.

Change into the `astropy` directory you created in the previous step and let git know about about the astropy remote:

```
cd astropy
git remote add astropy git://github.com/astropy/astropy.git
```

You can check that everything is set up properly so far by asking git to show you all of the remotes it knows about for your local repository of Astropy with `git remote -v`, which should display something like:

```
astropy   git://github.com/astropy/astropy.git (fetch)
astropy   git://github.com/astropy/astropy.git (push)
origin     git@github.com:your-user-name/astropy.git (fetch)
origin     git@github.com:your-user-name/astropy.git (push)
```

Note that git already knew about one remote, called *origin*; that is your fork of Astropy on GitHub.

To make more explicit that origin is really *your* fork of Astropy, rename that remote to your GitHub user name:

```
git remote rename origin your-user-name
```

Create your own private workspace

One of the nice things about git is that it is easy to make what is essentially your own private workspace to try out coding ideas. git calls these workspaces *branches*.

Your repository already has several branches; see them if you want by running `git branch -a`. Most of them are on `remotes/origin`; in other words, they exist on your remote copy of astropy on GitHub.

There is one special branch, called *master*. Right now it is the one you are working on; you can tell because it has a marker next to it in your list of branches: `* master`.

To make a long story short, you never want to work on master. Always work on a branch.

To avoid potential confusion down the road, make your own branch now; this one you can call anything you like (when making contributions you should use a meaningful more name):

```
git branch my-own-astropy
```

You are *not quite* done yet. Git knows about this new branch; run `git branch` and you get:

```
* master
  my-own-astropy
```

The `*` indicates you are still working on master. To work on your branch instead you need to *check out* the branch `my-own-astropy`. Do that with:

```
git checkout my-own-astropy
```

and you should be rewarded with:

```
Switched to branch 'my-own-astropy'
```

“Activate” the development version of astropy

Right now you have the development version of astropy, but python will not see it. Though there are more sophisticated ways of managing multiple versions of astropy, for now this straightforward way will work (if you want to jump ahead to the more sophisticated method look at Python virtual environments).

Note

If you want to work on C or Cython code in Astropy, this quick method of activating your copy of astropy will *not* work _ – you need to go straight to using a virtual python environment.

If you have decided to use the recommended “activation” method with `pip`, please note the following: Before trying to install, check that you have the required dependencies: “cython” and “jinja2”. If not, install them with `pip`. Note that on some platforms, the pip command is `pip3` instead of `pip`, so be sure to use this instead in the examples below if that is the case. If you have any problem with different versions of `pip` installed, try aliasing to resolve the issue. If you are unsure about which `pip` version you are using, try the command `which pip` on the terminal.

In the directory where your copy of astropy is type:

```
pip install -e .
```

Several pages of output will follow the first time you do this; this wouldn’t be a bad time to get a fresh cup of coffee. At the end of it you should see something like `Finished processing dependencies for astropy==3.2.dev6272`.

To make sure it has been activated **change to a different directory outside of the astropy distribution** and try this in python:

```
>>> import astropy
>>> astropy.__version__
'3.2.dev6272'
```

The actual version number will be different than in this example, but it should have `'dev'` in the name.

Warning

Right now every time you run Python, the development version of astropy will be used. That is fine for testing but you should make sure you change back to the stable version unless you are developing astropy. If you want to develop astropy, there is a better way of separating the development version from the version you do science with. That method, using a virtualenv, is discussed at Python virtual environments.

For now **remember to change back to your usual version** when you are done with this.

Test your development copy

Testing is an important part of making sure astropy produces reliable, reproducible results. Before you try out a new feature or think you have found a bug make sure the tests run properly on your system.

If the test *don't* complete successfully, that is itself a bug–please report it.

To run the tests, navigate back to the directory your copy of astropy is in on your computer, then, at the shell prompt, type:

```
pytest
```

This is another good time to get some coffee or tea. The number of test is large. When the test are done running you will see a message something like this:

```
4741 passed, 85 skipped, 11 xfailed
```

Skips and xfails are fine, but if there are errors or failures please report them.

Try out the development version

If you are going through this to ramp up to making more contributions to Astropy you don't actually have to do anything here.

If you are doing this because you have found a bug and are checking that it still exists in the development version, try running your code.

Or, just for fun, try out one of the new features in the development version.

Either way, once you are done, make sure you do the next step.

"Deactivate" the development version

Be sure to turn the development version off before you go back to doing science work with astropy.

Navigate to the directory where your local copy of the development version is, then run:

```
pip uninstall astropy
```

This should remove the development version only. Once again, it is important to check that you are using the proper version of `pip` corresponding to the Python executable desired.

You should really confirm it is deactivated by **changing to a different directory outside of the astropy distribution** and running this in python:

```
>>> import astropy
>>> astropy.__version__
'3.1.1'
```

The actual version number you see will likely be different than this example, but it should not have `'dev'` in it.

Developer Documentation

The developer documentation contains instructions for how to contribute to Astropy or affiliated packages, as well as coding, documentation, and testing guidelines. For the guiding vision of this process and the project as a whole, see Vision for a Common Astronomy Python Package.

How to make a code contribution

This document outlines the process for contributing code to the Astropy project.

Already experienced with git? Contributed before? Jump right to Astropy Guidelines for git.

Pre-requisites

Before following the steps in this document you need:

- an account on GitHub
- a local copy of the astropy source. Instructions for doing that, including the basics you need for setting up git and GitHub, are at Try the development version.

Strongly Recommended, but not required

You cannot easily work on the development version of astropy in a python environment in which you also use the stable version. It can be done — but can only be done *successfully* if you always remember whether the development version or stable version is the active one.

Python virtual environments offer a better solution and take only a few minutes to set up. It is well worth your time.

Not sure what your first contribution should be? Take a look at the Astropy issue list and grab one labeled "package-novice". These issues are the most accessible ones if you are not familiar with the Astropy source code. Issues labeled as "effort-low" are expected to take a few hours (at most) to address, while the "effort-medium" ones may take a few days. The developers are friendly and want you to help, so don't be shy about asking questions on the astropy-dev mailing list.

New to git?

Some git resources

If you have never used git or have limited experience with it, take a few minutes to look at these resources:

- Interactive tutorial that runs in a browser
- Git Basics, part of a much longer git book.

In practice, you need only a handful of git commands to make contributions to Astropy. There is a more extensive list of Git resources if you want more background.

Double check your setup

Before going further, make sure you have set up astropy as described in Try the development version.

In a terminal window, change directory to the one containing your clone of Astropy. Then, run `git remote`; the output should look something like this:

```
your-github-username
astropy
```

If that works, also run `git fetch --all`. If it runs without errors then your installation is working and you have a complete list of all branches in your clone, `your-github-username` and `astropy`.

About names in git

git is designed to be a *distributed* version control system. Each clone of a repository is, itself, a repository. That can lead to some confusion, especially for the branch called `master`. If you list all of the branches your clone of git knows about with `git branch -a` you will see there are *three* different branches called `master`:

```
* master                              # this is master in your local
repo
remotes/your-github-username/master   # master on your fork of
Astropy on GitHub
remotes/astropy/master                # the official development
branch of Astropy
```

The naming scheme used by git will also be used here. A plain branch name, like `master` means a branch in your local copy of Astropy. A branch on a remote, like `astropy` , is labeled by that remote, `astropy/master`.

This duplication of names can get very confusing when working with pull requests, especially when the official master branch, `astropy/master`, changes due to other contributions before your contributions are merged in. As a result, you should never do any work in your master branch, `master`. Always work on a branch instead.

Essential git commands

A full git tutorial is beyond the scope of this document but this list describes the few `git` commands you are likely to encounter in contributing to Astropy:

- `git fetch` gets the latest development version of Astropy, which you will use as the basis for making your changes.
- `git branch` makes a logically separate copy of Astropy to keep track of your changes.
- `git add` stages files you have changed or created for addition to git.
- `git commit` adds your staged changes to the repository.
- `git push` copies the changes you committed to GitHub
- `git status` to see a list of files that have been modified or created.

Note

A good graphical interface to git makes some of these steps much easier. Some options are described in Get a git GUI (optional).

If something goes wrong

git provides a number of ways to recover from errors. If you end up making a git mistake, do not hesitate to ask for help. An additional resource that walks you through recovering from git mistakes is the git choose-your-own-adventure.

Astropy Guidelines for git

- Don’t use your `master` branch for anything. Consider Deleting your master branch.
- Make a new branch, called a *feature branch*, for each separable set of changes: “one task, one branch” (ipython git workflow).
- Start that new *feature branch* from the most current development version of astropy (instructions are below).
- Name your branch for the purpose of the changes, for example `bugfix-for-issue-14` or `refactor-database-code`.
- Make frequent commits, and always include a commit message. Each commit should represent one logical set of changes.
- Ask on the astropy-dev mailing list if you get stuck.
- Never merge changes from `astropy/master` into your feature branch. If changes in the development version require changes to our code you can Rebase, but only if asked.

In addition there are a couple of git naming conventions used in this document:

- Change the name of the remote `origin` to `your-github-username`.
- Name the remote that is the primary Astropy repository `astropy`; in prior versions of this documentation it was referred to as `upstream`.

Workflow

These, conceptually, are the steps you will follow in contributing to Astropy:

1. Fetch the latest Astropy
2. Make a new feature branch; you will make your changes on this branch.
3. Install your branch
4. Follow The editing workflow to write/edit/document/test code - make frequent, small commits.
5. Add a changelog entry
6. Copy your changes to GitHub
7. From GitHub, Ask for your changes to be reviewed to let the Astropy maintainers know you have contributions to review.
8. Revise and push as necessary in response to comments on the pull request. Pushing those changes to GitHub automatically updates the pull request.

This way of working helps to keep work well organized, with readable history. This in turn makes it easier for project maintainers (that might be you) to see what you’ve done, and why you did it.

A worked example that follows these steps for fixing an Astropy issue is at Contributing code to Astropy, a worked example.

Some additional topics related to git are in Some other things you might want to do.

Deleting your master branch

It may sound strange, but deleting your own `master` branch can help reduce confusion about which branch you are on. See deleting master on github for details.

Fetch the latest Astropy

From time to time you should fetch the development version (i.e. Astropy `astropy/master`) changes from GitHub:

```
git fetch astropy --tags
```

This will pull down any commits you don't have, and set the remote branches to point to the latest commit. For example, 'trunk' is the branch referred to by `astropy/master`, and if there have been commits since you last checked, `astropy/master` will change after you do the fetch.

Make a new feature branch

Make the new branch

When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called 'feature branches'.

Making a new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. Branch names like `add-ability-to-fly` or `buxfix-for-issue-42` clearly describe the purpose of the branch.

Always make your branch from `astropy/master` so that you are basing your changes on the latest version of Astropy:

```
# Update the mirror of trunk
git fetch astropy --tags

# Make new feature branch starting at astropy/master
git branch my-new-feature astropy/master
git checkout my-new-feature
```

Connect the branch to GitHub

At this point you have made and checked out a new branch, but git does not know it should be connected to your fork on GitHub. You need that connection for your proposed changes to be managed by the Astropy maintainers on GitHub.

To connect your local branch to GitHub, you git push this new branch up to your GitHub repo with the `--set-upstream` option:

```
git push --set-upstream your-github-username my-new-feature
```

From now on git will know that `my-new-feature` is related to the `your-github-username/my-new-feature` branch in your GitHub fork of Astropy.

You will still need to `git push` your changes to GitHub periodically. The setup in this section will make that easier.

Install your branch

Ideally you should set up a Python virtual environment just for this fix; instructions for doing to are at Python virtual environments. Doing so ensures you will not corrupt your main `astropy` install and makes it very easy to recover from mistakes.

Once you have activated that environment, you need to install the version of `astropy` you are working on. Do that with:

```
pip install -e .
```

For more details on building `astropy` from source, see Building Astropy and its Subpackages.

The editing workflow

Conceptually, you will:

1. Make changes to one or more files and/or add a new file.
2. Check that your changes do not break existing code.
3. Add documentation to your code and, as appropriate, to the Astropy documentation.
4. Ideally, also make sure your changes do not break the documentation.
5. Add tests of the code you contribute.
6. Commit your changes in git
7. Repeat as necessary.

In more detail

1. Make some changes to one or more files. You should follow the Astropy Coding Guidelines. Each logical set of changes should be treated as one commit. For example, if you are fixing a known bug in Astropy and notice a different bug while implementing your fix, implement the fix to that new bug as a different set of changes.
2. Test that your changes do not lead to *regressions*, i.e. that your changes do not break existing code, by running the Astropy tests. You can run all of the Astropy tests from ipython with:

   ```
   import astropy
   astropy.test()
   ```

   If your change involves only a small part of Astropy, e.g. Time, you can run just those tests:

   ```
   import astropy
   astropy.test(package='time')
   ```

   Tests can also be run from the command line while in the package root directory, e.g.:

   ```
   pytest
   ```

   To run the tests in only a single package, e.g. Time, you can do:

   ```
   pytest -P time
   ```

   For more details on running tests, please see Testing Guidelines.
3. Make sure your code includes appropriate docstrings, in the Numpydoc format. If appropriate, as when you are adding a new feature, you should update the appropriate documentation in the `docs` directory; a detailed description is in Writing Documentation.
4. If you have sphinx installed, you can also check that the documentation builds and looks correct by running, from the `astropy` directory:

   ```
   cd docs
   make html
   ```

   The last line should just state `build succeeded`, and should not mention any warnings. (For more details, see Writing Documentation.)

5. Add tests of your new code, if appropriate. Some changes (e.g. to documentation) do not need tests. Detailed instructions are at Writing tests, but if you have no experience writing tests or with the pytest testing framework submit your changes without adding tests, but mention in the pull request that you have not written tests. An example of writing a test is in Stop and think: Any more tests or other changes?.
6. Stage your changes using `git add` and commit them using `git commit`. An example of doing that, based on the fix for an actual Astropy issue, is at Contributing code to Astropy, a worked example.

   Note

   Make your git commit messages short and descriptive. If a commit fixes an issue, include, on the second or later line of the commit message, the issue number in the commit message, like this: `Closes #123`. Doing so will automatically close the issue when the pull request is accepted.
7. Some modifications require more than one commit; if in doubt, break your changes into a few, smaller, commits rather than one large commit that does many things at once. Repeat the steps above as necessary!

Add a changelog entry

Add an entry to the file `CHANGES.rst` briefly describing the change you made. Include the pull request number, too at the end of the entry. An example entry, for the changes in PR 1845, is:

```
- ``astropy.wcs.Wcs.printwcs`` will no longer warn that ``cdelt`` is
  being ignored when none was present in the FITS file. [#1845]
```

If you are opening a new pull request, you may not know its number yet, but you can add it *after* you make the pull request. If you're not sure where to put the changelog entry, wait at least until a maintainer has reviewed your PR and assigned it to a milestone.

When writing changelog entries, do not attempt to make API reference links by using single-backticks. This is because the changelog (in its current format) runs for the history of the project, and API references you make today may not be valid in a future version of Astropy. However, use of double-backticks for monospace rendering of module/class/function/argument names and the like is encouraged.

Copy your changes to GitHub

This step is easy because of the way you created the feature branch. Just:

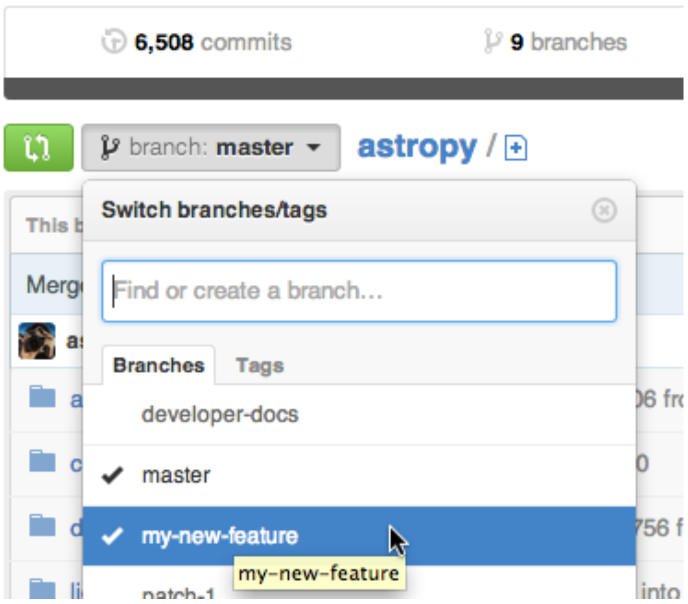
```
git push
```

Ask for your changes to be reviewed

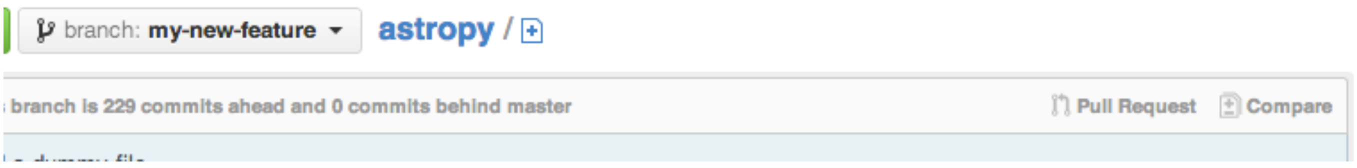
A *pull request* on GitHub is a request to merge the changes you have made into another repository.

When you are ready to ask for someone to review your code and consider merging it into Astropy:

1. Go to the URL of your fork of Astropy, e.g., `https://github.com/your-user-name/astropy`.
2. Use the ‘Switch Branches’ dropdown menu to select the branch with your changes:



3. Click on the ‘Pull request’ button:



Enter a title for the set of changes, and some explanation of what you’ve done. If there is anything you’d like particular attention for, like a complicated change or some code you are not happy with, add the details here.

If you don’t think your request is ready to be merged, just say so in your pull request message. This is still a good way to start a preliminary code review.

You may also opt to open a work-in-progress pull request. If you do so, instead of clicking “Create pull request”, click on the small down arrow next to it and select “Create draft pull request”. This will let the maintainers know that your work is not ready for a full review nor to be merged yet. In addition, if your commits are not ready for CI testing, you should also use `[ci skip]` or `[skip ci]` directive in your commit message.

Revise and push as necessary

You may be asked to make changes in the discussion of the pull request. Make those changes in your local copy, commit them to your local repo and push them to GitHub. GitHub will automatically update your pull request.

Do Not Create a Merge Commit

If your branch associated with the pull request falls behind the `master` branch of https://github.com/astropy/astropy, GitHub might offer you the option to catch up or resolve conflicts via its web interface, but do not use this. Using the web interface might create a “merge commit” in your commit history, which is undesirable, as a “merge commit” can introduce maintenance overhead for the release manager as well as undesirable branch structure complexity. Do not use the `git pull` command either.

Instead, in your local checkout, do a `fetch` and then a `rebase`, and resolve conflicts as necessary. See Rebase, but only if asked and How to rebase for further information.

Rebase, but only if asked

Sometimes the maintainers of Astropy may ask a pull request to be *rebased* or *squashed* in the process of reviewing a pull request for merging into the main Astropy *master* repository.

The decisions of when to request a *squash* or *rebase* are left to individual maintainers. These may be requested to reduce the number of visible commits saved in the repository history, or because of code changes in Astropy in the meantime. A rebase may be necessary to allow the Continuous Integration tests to run. Both involve rewriting the git history, meaning that commit hashes will change, which is why you should do it only if asked.

Conceptually, rebasing means taking your changes and applying them to the latest version of the development branch of the official Astropy as though that was the version you had originally branched from. Each individual commit remains visible, but with new metadata/commit hashes. Squashing commits changes the metadata/commit hash, and also removes separate visibility of individual commits; a new commit and commit message will only contain a textual list of the earlier commits.

It is easier to make mistakes rebasing than other areas of git, so before you start make a branch to serve as a backup copy of your work:

```
git branch tmp my-new-feature # make temporary branch--will be
deleted later
```

After altering the history, e.g. with `git rebase`, a normal `git push` is prevented, and a `git push --force` will be required.

How to rebase

Behind the scenes, git is deleting the changes and branch you made, making the changes others made to the development branch of Astropy, then re-making your branch from the development branch and applying your changes to your branch.

The actual rebasing is usually easy:

```
git fetch astropy master # get the latest development astropy
git rebase astropy/master my-new-feature
```

You are more likely to run into *conflicts* here — places where the changes you made conflict with changes that someone else made — than anywhere else. Ask for help if you need it. Instructions are available on how to resolve merge conflicts after a Git rebase.

How to squash

Typically we ask to *squash* when there was a fair amount of trial and error, but the final patch remains quite small, or when files were added and removed (especially binary files or files that should not remain in the repository) or if the number of commits in the history is disproportionate compared to the work being carried out (for example 30 commits gradually refining a final 10-line change). Conceptually this is equivalent to exporting the final diff from a feature branch, then starting a new branch and applying only that patch.

Many of us find that is it actually easiest to squash using rebase. In particular, you can rebase and squash within the existing branch using:

```
git fetch astropy
git rebase -i astropy/master
```

The last command will open an editor with all your commits, allowing you to squash several commits together, rename them, etc. Helpfully, the file you are editing has the instructions on what to do.

How to push

After using `git rebase` you will still need to push your changes to GitHub so that they are visible to others and the pull request can be updated. Use of a simple `git push` will be prevented because of the changed history, and will need to be manually overridden using:

```
git push --force
```

If you run into any problems, do not hesitate to ask. A more detailed conceptual discussing of rebasing is at Rebasing on trunk.

Once the modifications and new git history are successfully pushed to GitHub you can delete any backup branches that may have been created:

```
git branch -D tmp
```

When to rebase and squash commits

This page describes recommendations for when to rebase pull requests and when to combine/squash commits.

When to remove or combine/squash commits

Pull requests **must** be rebased and at least partially squashed (but not necessarily squashed to a single commit) if large (approximately >10KB) non-source code files (e.g. images, data files, etc.) are added and then removed or modified in the PR commit history (The squashing should remove all but the last addition of the file to not use extra space in the repository).

Combining/squashing commits is **encouraged** when the number of commits is excessive for the changes made. The definition of ‘excessive’ is subjective, but in general one should attempt to have individual commits be units of change, and not include reversions. As a concrete example, for a change affecting < 10 lines of source code and including a changelog entry, more than a few commits would be excessive. For a larger pull request adding significant functionality, however, more commits may well be appropriate.

As another guideline, squashing should remove extraneous information but should not be used to remove useful information for how a PR was developed. For example, 4 commits that are testing changes and have a commit message of just “debug” should be squashed. But a series of commit messages that are “Implemented feature X”, “added test for feature X”, “fixed bugs revealed by tests for feature X” are useful information and should not be squashed away without reason.

When squashing, extra care should be taken to keep authorship credit to all individuals who provided substantial contribution to the given PR, e.g. only squash commits made by the same author.

In all cases, be mindful of maintaining a welcoming environment and be helpful with advice, especially for new contributors. E.g., It is expected that a maintainer offer to help a contributor who is a novice git user do any squashing that that maintainer asks for, or do the squash themselves by directly pushing to the PR branch.

When to rebase

Pull requests **must** be rebased (but not necessarily squashed to a single commit) if at least one of the following conditions is met:

- There are conflicts with master
- There are merge commits from upstream/master in the PR commit history (merge commits from PRs to the user's fork are fine)
- There are commit messages include offensive language or violate the code of conduct (in this case the rebase must also edit the commit messages)

Github 'Squash and Merge' button

We should never use or enable the GitHub 'Squash and Merge' button since this creates problems when dealing with identifying backports.

Coding Guidelines

This section describes requirements and guidelines that should be followed both by the core package and by coordinated packages, and these are also recommended for affiliated packages.

Interface and Dependencies

- All code must be compatible with the versions of Python indicated by the `python_requires` key in the setup.cfg file of the core package.
- Usage of `six`, `__future__`, and `2to3` is no longer acceptable.
- f-strings should be used when possible, and if not, Python 3 formatting should be used (i.e. `"{0:s}".format("spam")`) instead of the `%` operator (`"%s" % "spam"`).
- The core package should be importable with no dependencies other than components already in the Astropy core, the Python Standard Library, and NumPy 1.17.0 or later.
- Additional dependencies - such as SciPy, Matplotlib, or other third-party packages - are allowed for sub-modules or in function calls, but they must be noted in the package documentation and should only affect the relevant component. In functions and methods, the optional dependency should use

a normal `import` statement, which will raise an `ImportError` if the dependency is not available. In the astropy core package, such optional dependencies should be recorded in the `setup.cfg` file in the `extras_require` entry.

At the module level, one can subclass a class from an optional dependency like so:

```
try:
    from opdep import Superclass
except ImportError:
    warn(AstropyWarning('opdep is not present, so <functionality>'
                        'will not work.'))
    class Superclass(object): pass

class Customclass(Superclass):
    ...
```

- General utilities necessary for but not specific to the package or sub-package should be placed in a `packagename.utils` module (e.g. `astropy.utils` for the core package). If a utility is already present in **`astropy.utils`**, packages should always use that utility instead of re-implementing it in `packagename.utils` module.

Documentation and Testing

- Docstrings must be present for all public classes/methods/functions, and must follow the form outlined in the Writing Documentation document.
- Write usage examples in the docstrings of all classes and functions whenever possible. These examples should be short and simple to reproduce–users should be able to copy them verbatim and run them. These examples should, whenever possible, be in the doctest format and will be executed as part of the test suite.
- Unit tests should be provided for as many public methods and functions as possible, and should adhere to the standards set in the Testing Guidelines document.

Data and Configuration

- Packages can include data in a directory named `data` inside a subpackage source directory as long as it is less than about 100 kB. These data should always be accessed via the **`get_pkg_data_fileobj()`** or **`get_pkg_data_filename()`** functions. If the data exceeds this size, it should be hosted outside the source code repository, either at a third-party

location on the internet or the astropy data server. In either case, it should always be downloaded using the **`get_pkg_data_fileobj()`** or **`get_pkg_data_filename()`** functions. If a specific version of a data file is needed, the hash mechanism described in **`astropy.utils.data`** should be used.
- All persistent configuration should use the Configuration System (astropy.config) mechanism. Such configuration items should be placed at the top of the module or package that makes use of them, and supply a description sufficient for users to understand what the setting changes.

Standard output, warnings, and errors

The built-in `print(...)` function should only be used for output that is explicitly requested by the user, for example `print_header(...)` or `list_catalogs(...)`. Any other standard output, warnings, and errors should follow these rules:

- For errors/exceptions, one should always use `raise` with one of the built-in exception classes, or a custom exception class. The nondescript `Exception` class should be avoided as much as possible, in favor of more specific exceptions (**`IOError`**, **`ValueError`**, etc.).
- For warnings, one should always use `warnings.warn(message, warning_class)`. These get redirected to `log.warning()` by default, but one can still use the standard warning-catching mechanism and custom warning classes. The warning class should be either **`AstropyUserWarning`** or inherit from it.
- For informational and debugging messages, one should always use `log.info(message)` and `log.debug(message)`.

The logging system uses the built-in Python **`logging`** module. The logger can be imported using:

```
from astropy import log
```

Coding Style/Conventions

- The code should follow the standard PEP8 Style Guide for Python Code. In particular, this includes using only 4 spaces for indentation, and never tabs.
- Our testing infrastructure currently enforces a subset of the PEP8 style guide. You can check locally whether your changes have followed these by running the following tox command:

```
tox -e codestyle
```

- *Follow the existing coding style* within a subpackage and avoid making changes that are purely stylistic. In particular, there is variation in the maximum line length for different subpackages (typically either 80 or 100 characters). Please try to maintain the style when adding or modifying code.
- The use of automatic code formatters (e.g., Black) is strongly discouraged in contributions to Astropy.
- Following PEP8's recommendation, absolute imports are to be used in general. The exception to this is relative imports of the form `from . import modname`, best when referring to files within the same sub-module. This makes it clearer what code is from the current submodule as opposed to from another.

  Note

  There are multiple options for testing PEP8 compliance of code, see Testing Guidelines for more information. See Emacs setup for following coding guidelines for some configuration options for Emacs that helps in ensuring conformance to PEP8.

- Astropy source code should contain a comment at the beginning of the file (or immediately after the `#!/usr/bin env python` command, if relevant) pointing to the license for the Astropy source code. This line should say:

```
# Licensed under a 3-clause BSD style license - see LICENSE.rst
```

- The following naming conventions:

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
```

  should be used wherever relevant. On the other hand:

```
from packagename import *
```

  should never be used, except as a tool to flatten the namespace of a module. An example of the allowed usage is given in the Acceptable use of from module import * example.

- Classes should either use direct variable access, or Python's property mechanism for setting object instance variables. `get_value` / `set_value` style methods should be used only when getting and setting the values requires a computationally-expensive operation. The Properties vs. get_/set_ example below illustrates this guideline.

- Classes should use the builtin **`super()`** function when making calls to methods in their super-class(es) unless there are specific reasons not to. **`super()`** should be used consistently in all subclasses since it does not work otherwise. The super() vs. Direct Calling example below illustrates why this is important.
- Multiple inheritance should be avoided in general without good reason. Multiple inheritance is complicated to implement well, which is why many object-oriented languages, like Java, do not allow it at all. Python does enable multiple inheritance through use of the C3 Linearization algorithm, which provides a consistent method resolution ordering. Non-trivial multiple-inheritance schemes should not be attempted without good justification, or without understanding how C3 is used to determine method resolution order. However, trivial multiple inheritance using orthogonal base classes, known as the 'mixin' pattern, may be used.
- `__init__.py` files for modules should not contain any significant implementation code. `__init__.py` can contain docstrings and code for organizing the module layout, however (e.g. `from submodule import *` in accord with the guideline above). If a module is small enough that it fits in one file, it should simply be a single file, rather than a directory with an `__init__.py` file.
- Command-line scripts should follow the form outlined in the Writing Command-Line Scripts document.

Unicode guidelines

For maximum compatibility, we need to assume that writing non-ASCII characters to the console or to files will not work. However, for those that have a correctly configured Unicode environment, we should allow them to opt-in to take advantage of Unicode output when appropriate. Therefore, there is a global configuration option, `astropy.conf.unicode_output` to enable Unicode output of values, set to **`False`** by default.

The following conventions should be used for classes that define the standard string conversion methods (`__str__`, `__repr__`, `__bytes__`, and `__format__`). In the bullets below, the phrase "string instance" is used to refer to **`str`**, while "bytes instance" is used to refer to **`bytes`**.

- `__repr__`: Return a "string instance" containing only 7-bit characters.
- `__bytes__`: Return a "bytes instance" containing only 7-bit characters.
- `__str__`: Return a "string instance". If

`astropy.conf.unicode_output` is **False**, it must contain only 7-bit characters. If `astropy.conf.unicode_output` is **True**, it may contain non-ASCII characters when applicable.

- `__format__`: Return a “string instance”. If `astropy.conf.unicode_output` is **False**, it must contain only 7-bit characters. If `astropy.conf.unicode_output` is **True**, it may contain non-ASCII characters when applicable.

For classes that are expected to roundtrip through strings (unicode or bytes), the parser must accept the output of `__str__`. Additionally, `__repr__` should roundtrip when that makes sense.

This design generally follows Postel’s Law: “Be liberal in what you accept, and conservative in what you send.”

The following example class shows a way to implement this:

```
# -*- coding: utf-8 -*-

from astropy import conf

class FloatList(object):
    def __init__(self, init):
        if isinstance(init, str):
            init = init.split('‖')
        elif isinstance(init, bytes):
            init = init.split(b'|')
        self.x = [float(x) for x in init]

    def __repr__(self):
        # Return unicode object containing no non-ASCII characters
        return '<FloatList [{0}]>'.format(', '.join(
            str(x) for x in self.x))

    def __bytes__(self):
        return b'|'.join(bytes(x) for x in self.x)

    def __str__(self):
        if astropy.conf.unicode_output:
            return '‖'.join(str(x) for x in self.x)
        else:
            return self.__bytes__().decode('ascii')
```

Additionally, there is a test helper, `astropy.test.helper.assert_follows_unicode_guidelines` to ensure that a class follows the Unicode guidelines outlined above. The following example test will test that our example class above is compliant:

```
def test_unicode_guidelines():
    from astropy.test.helper import assert_follows_unicode_guidelines
    assert_follows_unicode_guidelines(FloatList(b'5|4|3|2'),
roundtrip=True)
```

Including C Code

- C extensions are only allowed when they provide a significant performance enhancement over pure Python, or a robust C library already exists to provided the needed functionality. When C extensions are used, the Python interface must meet the aforementioned Python interface guidelines.
- The use of Cython is strongly recommended for C extensions. Cython extensions should store `.pyx` files in the source code repository, but not the generated `.c` files.
- If a C extension has a dependency on an external C library, the source code for the library should be bundled with the Astropy core, provided the license for the C library is compatible with the Astropy license. Additionally, the package must be compatible with using a system-installed library in place of the library included in Astropy, and a user installing the package should be able to opt-in to using the system version using a `ASTROPY_USE_SYSTEM_???` environment variable, where `???` is the name of the library, e.g. `ASTROPY_USE_SYSTEM_WCSLIB` (see also External C Libraries).
- In cases where C extensions are needed but Cython cannot be used, the PEP 7 Style Guide for C Code is recommended.
- C extensions (Cython or otherwise) should provide the necessary information for building the extension via the mechanisms described in C or Cython Extensions.

Requirements Specific to Affiliated Packages

- Affiliated packages implementing many classes/functions not relevant to the affiliated package itself (for example leftover code from a previous package) will not be accepted - the package should only include the required functionality and relevant extensions.
- Affiliated packages must be registered on the Python Package Index, with proper metadata for downloading and installing the source package.
- The `astropy` root package name should not be used by affiliated packages - it is reserved for use by the core package.

Examples

This section shows a few examples (not all of which are correct!) to illustrate points from the guidelines.

Properties vs. get_/set_

This example shows a sample class illustrating the guideline regarding the use of properties as opposed to getter/setter methods.

Let's assuming you've defined a `Star` class and create an instance like this:

```
>>> s = Star(B=5.48, V=4.83)
```

You should always use attribute syntax like this:

```
>>> s.color = 0.4
>>> print(s.color)
0.4
```

Rather than like this:

```
>>> s.set_color(0.4)  # Bad form!
>>> print(s.get_color())  # Bad form!
0.4
```

Using Python properties, attribute syntax can still do anything possible with a get/set method. For lengthy or complex calculations, however, use a method:

```
>>> print(s.compute_color(5800, age=5e9))
0.4
```

super() vs. Direct Calling

This example shows why the use of **`super()`** leads to a more consistent method resolution order than manually calling methods of the super classes in a multiple inheritance case:

```
# This is dangerous and bug-prone!

class A(object):
    def method(self):
        print('Doing A')


class B(A):
    def method(self):
        print('Doing B')
        A.method(self)
```

```
class C(A):
    def method(self):
        print('Doing C')
        A.method(self)

class D(C, B):
    def method(self):
        print('Doing D')
        C.method(self)
        B.method(self)
```

if you then do:

```
>>> b = B()
>>> b.method()
```

you will see:

```
Doing B
Doing A
```

which is what you expect, and similarly for C. However, if you do:

```
>>> d = D()
>>> d.method()
```

you might expect to see the methods called in the order D, B, C, A but instead you see:

```
Doing D
Doing C
Doing A
Doing B
Doing A
```

because both `B.method()` and `C.method()` call `A.method()` unaware of the fact that they're being called as part of a chain in a hierarchy. When `C.method()` is called it is unaware that it's being called from a subclass that inherits from both `B` and `C`, and that `B.method()` should be called next. By calling **`super()`** the entire method resolution order for `D` is precomputed, enabling each superclass to cooperatively determine which class should be handed control in the next **`super()`** call:

```
# This is safer
```

```
class A(object):
    def method(self):
        print('Doing A')

class B(A):
    def method(self):
        print('Doing B')
        super().method()


class C(A):
    def method(self):
        print('Doing C')
        super().method()

class D(C, B):
    def method(self):
        print('Doing D')
        super().method()
```

```
>>> d = D()
>>> d.method()
Doing D
Doing C
Doing B
Doing A
```

As you can see, each superclass's method is entered only once. For this to work it is very important that each method in a class that calls its superclass's version of that method use **`super()`** instead of calling the method directly. In the most common case of single-inheritance, using `super()` is functionally equivalent to calling the superclass's method directly. But as soon as a class is used in a multiple-inheritance hierarchy it must use `super()` in order to cooperate with other classes in the hierarchy.

Note

For more information on the the benefits of **`super()`**, see https://rhettinger.wordpress.com/2011/05/26/super-considered-super/

Acceptable use of `from module import *`

`from module import *` is discouraged in a module that contains implementation code, as it impedes clarity and often imports unused variables. It can, however, be used for a package that is laid out in the following manner:

```
packagename
packagename/__init__.py
packagename/submodule1.py
packagename/submodule2.py
```

In this case, `packagename/__init__.py` may be:

```
"""
A docstring describing the package goes here
"""
from submodule1 import *
from submodule2 import *
```

This allows functions or classes in the submodules to be used directly as `packagename.foo` rather than `packagename.submodule1.foo`. If this is used, it is strongly recommended that the submodules make use of the `__all__` variable to specify which modules should be imported. Thus, `submodule2.py` might read:

```
from numpy import array, linspace

__all__ = ['foo', 'AClass']

def foo(bar):
    # the function would be defined here
    pass

class AClass(object):
    # the class is defined here
    pass
```

This ensures that `from submodule import *` only imports `foo` and `AClass`, but not **`numpy.array`** or **`numpy.linspace`**.

Additional Resources

Further tips and hints relating to the coding guidelines are included below.

Emacs setup for following coding guidelines

The Astropy coding guidelines are listed in Coding Guidelines. Here, we describe how to configure Emacs to help ensure Python code satisfies the guidelines.

For this setup, we add to the standard `python-mode` using flycheck and the flake8 python style checker. For installation instructions, see their respective web sites (or install via your distribution; e.g., in Debian/Ubuntu, the packages are called `elpa-flycheck` and `flake8`).

Note

Emacs can be configured in several different ways. So instead of providing a drop in configuration file, only the individual configurations are presented below.

The setup below is on purpose minimal. In principle, it is possible to use Emacs for Python development, with, e.g., elpy.

No tabs

This setting will cause indentation to use spaces rather than tabs for all files. For python files, indentation of 4 spaces will be used if the tab key is pressed.

```
;; Don't use TABS for indentations.
(setq-default indent-tabs-mode nil)
```

Delete trailing white spaces

One can delete trailing whitespace with `M-x delete-trailing-whitespace`. To ensure this is done every time a python file is saved, use:

```
;; Automatically remove trailing whitespace when file is saved.
(add-hook 'python-mode-hook
(lambda () (add-to-list 'write-file-functions 'delete-trailing-
whitespace)))
```

If you want to use this for every type of file, you can use `(add-hook 'before-save-hook 'delete-trailing-whitespace)`.

Flycheck

One can make lines that do not satisfy syntax requirements using flycheck. When the cursor is on such a line a message is displayed in the mini-buffer. When mouse pointer is on such a line a “tool tip” message is also shown. By default, flycheck will check if flake8 is installed and, if so, use that for its syntax checking. To ensure flycheck starts upon opening python files, add:

```
(add-hook 'python-mode-hook 'flycheck-mode)
```

Alternatively, you can just use `(global-flycheck-mode)` to run flycheck for all languages it supports.

Writing Documentation

High-quality, consistent documentation for astronomy code is one of the major goals of the Astropy Project. Hence, we describe our documentation procedures and rules here. For the astropy core project and coordinated packages we try to keep to these as closely as possible, and we encourage affiliated packages to also adhere to these as they encourage useful documentation, a characteristic often lacking in professional astronomy software.

Adding a Git Commit

When your changes only affect documentation (i.e., docstring or RST files) and do not include any code snippets that require doctest to run, you may add a `[ci skip]` in your commit message. For example:

```
git commit -m "Update documentation about this and that [ci skip]"
```

When this commit is pushed out to your branch associated with a pull request, all CI will be skipped because it is not required. This is because the the documentation build resides in RTD, which currently does not respect the `[ci skip]` directive.

Building the Documentation from source

For information about building the documentation from source, see the Building Documentation section in the installation instructions.

Astropy Documentation Rules and Guidelines

This section describes the standards for documentation that any contribution being considered for integration into the core package should follow, as well as the standard Astropy docstring format.

- All documentation text should follow the Astropy Narrative Style Guide: A Writing Resource for Contributors.
- All documentation should be written use the Sphinx documentation tool.
- The package template provides a recommended general structure for documentation.
- Docstrings must be provided for all public classes, methods, and functions.
- Docstrings should follow the numpydoc format.
- Examples and/or tutorials are strongly encouraged for typical use-cases of a particular module or class.
- Any external package dependencies must be explicitly mentioned in the documentation. They should also be recorded in the `setup.cfg` file in the root of the astropy repository using an `extras_require` entry.
- Configuration options using the **`astropy.config`** mechanisms must be

explicitly mentioned in the documentation.

Sphinx Documentation Themes

An Astropy Project Sphinx HTML theme is included in the astropy-sphinx-theme package. This allows the theme to be used by both Astropy and affiliated packages. The theme is activated by setting the theme in the global Astropy sphinx configuration in sphinx-astropy, which is imported in the sphinx configuration of both Astropy and affiliated packages.

A different theme can be used by overriding a few sphinx configuration variables set in the global configuration.

- To use a different theme, set `html_theme` to the name of a desired builtin Sphinx theme or a custom theme in `package-name/docs/conf.py` (where `'package-name'` is “astropy” or the name of the affiliated package).
- To use a custom theme, additionally: place the theme in `package-name/docs/_themes` and add `'_themes'` to the `html_theme_path` variable. See the Sphinx documentation for more details on theming.

Sphinx extensions

The documentation build process for Astropy uses a number of sphinx extensions which are all installed automatically when installing sphinx-astropy. These facilitate easily documenting code in a homogeneous and readable way.

The main extensions used are:

- sphinx-automodapi - an extension that makes it easy to automatically generate API documentation.
- sphinx-gallery - an extension to generate example galleries
- numpydoc - an extension to parse docstrings in NumpyDoc format

In addition, the sphinx-astropy includes a few small extensions:

- `sphinx_astropy.ext.edit_on_github` - an extension to add ‘Edit on GitHub’ links to documentation pages.
- `sphinx_astropy.ext.changelog_links` - an extension to add links to pull requests when rendering the changelog.
- `sphinx_astropy.ext.doctest` - an extension that makes it possible to add metadata about doctests inside `.rst` files

Astropy Narrative Style Guide: A Writing Resource for Contributors

The purpose of this style guide is to provide the Astropy community with a set of style and formatting guidelines that can be referenced when writing Astropy documentation. Following the guidelines offered in this style guide will bring greater consistency and clarity to Astropy's documentation, supporting its mission to develop a common core package for Astronomy in Python and foster an ecosystem of interoperable astronomy packages.

This style guide is organized alphabetically by writing topic, with usage examples in each section, and tone and formatting guidelines at the end.

Abbreviations

Place abbreviations such as i.e. and e.g. within parentheses, where they are followed by a comma. Alternatively, consider using "that is" and "for example" instead, preceded by an em dash or semicolon and followed by a comma, or contained within em dashes.

Examples

- The only way to modify the data in a frame is by using the `data` attribute directly and not the aliases for components on the frame (i.e., the following will not work).
- There are no plans to support more complex evolution (e.g., non-inertial frames or more complex evolution), as that is out of scope for the `astropy` core.
- Once you have a coordinate object you can access the components of that coordinate — for example, RA or Dec — to get string representations of the full coordinate.

For general use and scientific terms, use abbreviations only when the abbreviated term is well-known and widely used within the astronomy community. For less common scientific terms, or terms specific to a given field, write out the term or link to a resource of explanation. A good rule of thumb to follow when deciding whether or not something should be abbreviated is: when in doubt, write it out.

Examples

- 1D, 2D, etc. is preferred over one-dimensional, two-dimensional, etc.
- Units such as SI and CGS can be abbreviated as is more commonly seen in the scientific community.
- White dwarf should be written out fully instead of abbreviated as WD.
- Names of organizations or other proper nouns that employ acronyms should be written as their known acronym, but with a hyperlink to a website or resource for reference, for instance, CODATA.

Capitalization

Capitalize all proper nouns (names) in plain text, except when referring to package/code names, in which case use lowercase and double backticks. Astropy capitalized refers to The Astropy Project, while `astropy` lowercase and in backticks refers to the core package.

Examples

- Follow Astropy guidelines for contributing code.
- Affiliated packages are astronomy-related software packages that are not part of the `astropy` core package.
- Provide a code example along with details of the operating system and the Python, `numpy`, and `astropy` versions you are using.

In Documentation materials, title case capitalization is preferred in headings, meaning capitalize first, last, and all major words in the heading, but lowercase articles (the, a, an), prepositions (at, to, up, down, with, in, etc.), and common coordinating conjunctions (and, but, for, or). Sentence case capitalization is acceptable for longer example headings.

Examples

- Building and Installing
- Frames without Data
- Checklist for Contributing Code
- Astropy Guidelines
- Importing `astropy` and Subpackages
- Example: Use velocity to compute sky position at different epochs

In Tutorials and other learning materials, title case capitalization is preferred in headings of structured introductory/template sections, but within the tutorial, sentence case (i.e., capitalize first word and proper nouns only) is acceptable for longer headings designating different learning/code sections.

Contractions

Do not use contractions in formal documentation material.

Examples

- If you are making changes that impact `astropy` performance, consider adding a performance benchmark.
- You do not need to include a changelog entry.

In all other materials, avoid use of contractions only when the tense can be confused, such as in the case of “she is gone” versus “she has gone,” etc.

Hyphenation

Phrasal adjectives/compound modifiers placed before a noun should be hyphenated to avoid confusion.

Examples

- Astronomy-related software packages.
- Astropy provides sustainable, high-level education to the astronomy community.

Hyphenated compound words should contain hyphens in plain text, but no hyphens in code.

Example

- Do not forget to double-check your formatting.

Numbers

For numbers followed by a unit or as part of a name, use the numeral.

Examples

- 1 arcminute
- 32 degrees
- Gaia data release 2 catalog
- 1D, 2D, etc. is preferred over one-dimensional, two-dimensional, etc.

For all other whole numbers, follow Associated Press (AP) style: spell out numbers one through nine, and use numerals for 10 and higher, with numeral-word combinations for millions, billions, and trillions.

Examples

- There are two ways to build Astropy documentation.
- Follow these 11 steps.
- Measuring astrometry for about 2 billion stars.

For casual expressions, spell out the number.

Example

- A picture is worth a thousand words.

Punctuation

For consistency across Astropy materials, non-U.S. punctuation will be edited to reflect American punctuation preferences.

Parentheses: punctuation belonging to parenthetical material will be placed inside of closing parentheses, with the exception of commas to denote a small pause coming after parenthetical material, and periods when parenthetical material is included within another sentence.

Examples

- (For full contributor guidelines, see our documentation.)
- Once you open a pull request (which should be opened against the `master` branch), please make sure to include the following.
- In some cases, most of the required functionality is contained in a single class (or a few classes).

Quotation marks: periods and commas will be placed inside of closing quotation marks, whether double or single.

Examples

- Chief among these terms is the concept of a “coordinate system.”
- Because of the likelihood of confusion between these meanings of “coordinate system,” **`coordinates`** avoids this term wherever possible.

Hyphens vs. En Dashes vs. Em Dashes

Hyphens (-) should be used for phrasal adjectives and compound words (see Hyphenation above).

En dashes (– longer) should be used for number ranges (dates, times, pages) or to replace the words “to” or “through,” without spaces around the dash.

Examples

- See chapters 14–18.
- We have blocked off March 2019–May 2019 to develop a new version.

Em dashes (— longest) can be used in place of commas, parentheses, or colons to set off amplifying or explanatory elements. In Astropy materials, follow Associated Press (AP) style, which calls for spaces on either side of each em dash.

Examples

- Several types of input angles — array, scalar, tuple, string — can be used in

the creation of an Angle object.
- The creation of an Angle object supports a variety of input angle types — array, scalar, tuple, string, etc.

Spelling

For consistency across Astropy materials, non-U.S. spelling will be edited to reflect American spelling preferences.

Example

- Cross-matching catalog coordinates (versus catalogue)

Time and Date

Use numerals when exact times are expressed. Use the 24-hour system to express exact times. For consistency across Astropy materials, all instances of exact times will be edited to reflect 24-hour time system preferences.

Example

- The presentation starts at 15:00.

Express specific dates as numerals in ISO 8601 format, year-month-day.

Example

- Data from the Gaia mission was released on 2018-04-25.

A Note About Voice and Tone

Across all Astropy materials in narrative sections, please follow these voice and tone guidelines.

Write in the present tense.

Example

- In the following section, we are going to make a plot…
- To test if your version of `astropy` is running correctly…

Use the first-person inclusive plural.

Example

- We did this the long way, but next we can try it the short way…

Use the generic pronoun “you” instead of “one.”

Example

- You can access any of the attributes on a frame by…

Always avoid extraneous or belittling words such as “obviously,” “easily,” “simply,” “just,” or “straightforward.” Avoid extraneous phrases like, “we just have to do one more thing.”

Avoid words or phrases that create worry in the mind of the reader. Instead, use positive language that establishes confidence in the skills being learned.

Examples

- As a best practice…
- One recommended way to…
- An important note to remember is…

Along these lines, use “warning” directives only to note limitations in the code, not implied limitations in the skills or knowledge of the reader.

Documentation vs. Tutorials vs. Guides

Documentation

Tone: academic and slightly more formal.

- Use title case capitalization in section headings.
- Do not use contractions.

Tutorials

Tone: academic but less formal and more friendly.

- Use title case capitalization in introductory/template headings, switch to sentence case capitalization for learning/example section headings.
- Section headings should use the imperative mood to form a command or request (e.g., “Download the data”).
- Contractions can be used as long as the tense is clear.

Guides

Tone: academic but less formal and more friendly.

- Use title case capitalization in introductory/template headings, switch to sentence case capitalization for learning/example section headings.
- Contractions can be used as long as the tense is clear.

Formatting Guidelines

Astropy documentation is written in reStructuredText using the Sphinx documentation generator. When formatting the different sections of your documentation files, please follow these guidelines to maintain consistency in section heading hierarchy across Astropy's RST files.

Section headings in reStructuredText files are created by underlining (and optionally overlining) the section title with a punctuation character the same length as the text.

Examples

```
*************************
This is a Chapter Heading
*************************
```

```
This is a Section Heading
=========================
```

Although there are no formally assigned characters to create heading level hierarchy, as the hierarchy rendering is determined from the succession of headings, here is a suggested convention to follow when formatting Astropy documentation files:

# with overline, for parts * with overline, for chapters =, for sections -, for subsections ^, for subsubsections “, for paragraphs

These guidelines follow Sphinx's recommendation in the Sections chapter of its reStructuredText Primer and Python's convention in the 7.3.6. Sections part of its style guide.

Other Writing Resources

Some other resources that may be useful when writing Astropy documentation are:

- Python's Style Guide
- Sphinx's reStructuredText Primer
- Quick reStructuredText

Testing Guidelines

This section describes the testing framework and format standards for tests in Astropy core and coordinated packages, and also serves as recommendations for affiliated packages.

Testing Framework

The testing framework used by astropy (and packages using the Astropy package template) is the pytest framework.

Testing Dependencies

The dependencies used by the Astropy test runner are provided by a separate package called pytest-astropy. This package provides the `pytest` dependency itself, in addition to several `pytest` plugins that are used by Astropy, and will also be of general use to other packages.

Since the testing dependencies are not actually required to install or use Astropy, they are not included in `install_requires` in `setup.cfg`. Instead, they are listed in an `extras_require` section called `test` in `setup.cfg`. Developers who want to run the test suite will need to either install pytest-astropy directly:

```
pip install pytest-astropy
```

or install the core package in ‘editable’ mode specifying the `[test]` option:

```
pip install -e .[test]
```

A detailed description of the plugins can be found in the Pytest Plugins section.

Running Tests

There are currently three different ways to invoke Astropy tests. Each method invokes pytest to run the tests but offers different options when calling. To run the tests, you will need to make sure you have the pytest package installed.

In addition to running the Astropy tests, these methods can also be called so that they check Python source code for PEP8 compliance. All of the PEP8 testing options require the pytest-pep8 plugin, which must be installed separately.

tox

The most robust way to run the tests (which can also be the slowest) is to make use of Tox, which is a general purpose tool for automating Python testing. One of the benefits of tox is that it first creates a source distribution of the package being tested, and installs it into a new virtual environment, along with any dependencies that are declared in the package, before running the tests. This can therefore catch issues related to undeclared package data, or missing dependencies. Since we use tox to run many of the tests on continuous integration services, it can also be used in many cases to reproduce issues seen on those services.

To run the tests with tox, first make sure that tox is installed, e.g.:

```
pip install tox
```

then run the basic test suite with:

```
tox -e test
```

or run the test suite with all optional dependencies with:

```
tox -e test-alldeps
```

You can see a list of available test environments with:

```
tox -l -v
```

which will also explain what each of them does.

You can also run checks or commands not directly related to tests - for instance:

```
tox -e codestyle
```

will run checks using the flake8 tool.

Is is possible to pass options to pytest when running tox - to do this, add a `--` after the regular tox command, and anything after this will be passed to pytest, e.g.:

```
tox -e test -- -v --pdb
```

This can be used in conjunction with the `-P` option provided by the pytest-filter-subpackage plugin to run just part of the test suite.

pytest

The test suite can also be run directly from the native `pytest` command, which is generally faster than using tox for iterative development. In this case, it is important for developers to be aware that they must manually rebuild any extensions by running:

```
pip install -e .[test]
```

before running the test with pytest with:

```
pytest
```

Instead of calling `pip install -e .[test]`, you can also build the extensions with:

```
python setup.py build_ext --inplace
```

which avoids also installing the developer version of astropy into your current environment - however note that the `pip` command is required if you need to test parts of the package that rely on certain entry points being installed.

It is possible to run only the tests for a particular subpackage or set of subpackages. For example, to run only the `wcs` tests from the commandline:

```
pytest -P wcs
```

Or, to run only the `wcs` and `utils` tests:

```
pytest -P wcs,utils
```

You can also specify a single directory or file to test from the commandline, e.g.:

```
pytest astropy/modeling
```

or:

```
pytest astropy/wcs/tests/test_wcs.py
```

and this works for `.rst` files too:

```
pytest astropy/wcs/index.rst
```

astropy.test()

Tests can be run from an installed version of Astropy with:

```
import astropy
astropy.test()
```

This will run all the default tests for Astropy (but will not run the documentation tests in the `.rst` documentation since those files are not installed).

Tests for a specific package can be run by specifying the package in the call to the `test()` function:

```
astropy.test(package='io.fits')
```

This method works only with package names that can be mapped to Astropy directories. As an alternative you can test a specific directory or file with the `test_path` option:

```
astropy.test(test_path='wcs/tests/test_wcs.py')
```

The `test_path` must be specified either relative to the working directory or absolutely.

By default astropy.test() will skip tests which retrieve data from the internet. To turn these tests on use the `remote_data` flag:

```
astropy.test(package='io.fits', remote_data=True)
```

In addition, the `test` function supports any of the options that can be passed to pytest.main() and convenience options `verbose=` and `pastebin=`.

Enable PEP8 compliance testing with `pep8=True` in the call to `astropy.test`. This will enable PEP8 checking and disable regular tests.

Astropy Test Function

`astropy.`**`test`**(***kwargs*)

Run the tests for the package.

This method builds arguments for and then calls `pytest.main`.

Parameters: **package** : *str, optional*

The name of a specific package to test, e.g. 'io.fits' or 'utils'. Accepts comma separated string to specify multiple packages. If nothing is specified all default tests are run.

args : *str, optional*

Additional arguments to be passed to `pytest.main` in the `args` keyword argument.

docs_path : *str, optional*

The path to the documentation .rst files.

open_files : *bool, optional*

Fail when any tests leave files open. Off by default, because this adds extra run time to the test suite. Requires the `psutil` package.

parallel : *int or 'auto', optional*

When provided, run the tests in parallel on the specified number of CPUs. If parallel is `'auto'`, it will use the all the cores on the machine. Requires the `pytest-xdist` plugin.

pastebin : *('failed', 'all', None), optional*

Convenience option for turning on pytest pastebin output. Set to 'failed' to upload info for failed tests, or 'all' to upload info for all tests.

pdb : *bool, optional*

Turn on PDB post-mortem analysis for failing tests. Same as specifying `--pdb` in `args`.

pep8 : *bool, optional*

Turn on PEP8 checking via the pytest-pep8 plugin and disable normal tests. Same as specifying `--pep8 -k pep8` in `args`.

plugins : *list, optional*

Plugins to be passed to `pytest.main` in the `plugins` keyword argument.

remote_data : *{'none', 'astropy', 'any'}, optional*

Controls whether to run tests marked with @pytest.mark.remote_data. This can be set to run no tests with remote data (`none`), only ones that use data from http://data.astropy.org (`astropy`), or all tests that use remote data (`any`). The default is `none`.

Test-running options

Testing for open files

Using the pytest-openfiles plugin (which is installed automatically when installing pytest-astropy), we can test whether any of the unit tests inadvertently leave any files open. Since this greatly slows down the time it takes to run the tests, it is turned off by default.

To use it from the commandline, do:

```
pytest --open-files
```

To use it from Python, do:

```
>>> import astropy
>>> astropy.test(open_files=True)
```

For more information on the `pytest-openfiles` plugin see pytest-openfiles

Test coverage reports

Coverage reports can be generated using the pytest-cov plugin (which is installed automatically when installing pytest-astropy) by using e.g.:

```
pytest --cov astropy --cov-report html
```

There is some configuration inside the `setup.cfg` file that defines files to omit as well as lines to exclude.

Running tests in parallel

It is possible to speed up astropy's tests using the pytest-xdist plugin.

Once installed, tests can be run in parallel using the `'-n'` commandline option. For example, to use 4 processes:

```
pytest -n 4
```

Pass `-n auto` to create the same number of processes as cores on your machine.

Similarly, this feature can be invoked from `astropy.test`:

```
>>> import astropy
>>> astropy.test(parallel=4)
```

Writing tests

`pytest` has the following test discovery rules:

- `test_*.py` or `*_test.py` files
- `Test` prefixed classes (without an `__init__` method)
- `test_` prefixed functions and methods

Consult the test discovery rules for detailed information on how to name files and tests so that they are automatically discovered by pytest.

Simple example

The following example shows a simple function and a test to test this function:

```
def func(x):
    """Add one to the argument."""
    return x + 1

def test_answer():
    """Check the return value of func() for an example argument."""
    assert func(3) == 5
```

If we place this in a `test.py` file and then run:

```
pytest test.py
```

The result is:

```
============================= test session starts
==============================
python: platform darwin -- Python 3.x.x -- pytest-x.x.x
test object 1: /Users/username/tmp/test.py
```

```
test.py F

=================================== FAILURES
===================================
_________________________________ test_answer
__________________________________

    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)

test.py:5: AssertionError
=========================== 1 failed in 0.07 seconds
===========================
```

Where to put tests

Package-specific tests

Each package should include a suite of unit tests, covering as many of the public methods/functions as possible. These tests should be included inside each sub-package, e.g:

```
astropy/io/fits/tests/
```

`tests` directories should contain an `__init__.py` file so that the tests can be imported and so that they can use relative imports.

Interoperability tests

Tests involving two or more sub-packages should be included in:

```
astropy/tests/
```

Regression tests

Any time a bug is fixed, and wherever possible, one or more regression tests should be added to ensure that the bug is not introduced in future. Regression tests should include the ticket URL where the bug was reported.

Working with data files

Tests that need to make use of a data file should use the **`get_pkg_data_fileobj`** or **`get_pkg_data_filename`** functions. These functions search locally first, and then on the astropy data server or an arbitrary URL, and return a file-like object or a local filename, respectively. They automatically cache the data locally if remote data is obtained, and from then on the local copy will be used transparently. See the next section for note specific to dealing with the cache in tests.

They also support the use of an MD5 hash to get a specific version of a data file. This hash can be obtained prior to submitting a file to the astropy data server by using the **`compute_hash`** function on a local copy of the file.

Tests that may retrieve remote data should be marked with the `@pytest.mark.remote_data` decorator, or, if a doctest, flagged with the `REMOTE_DATA` flag. Tests marked in this way will be skipped by default by `astropy.test()` to prevent test runs from taking too long. These tests can be run by `astropy.test()` by adding the `remote_data='any'` flag. Turn on the remote data tests at the command line with `pytest --remote-data=any`.

It is possible to mark tests using `@pytest.mark.remote_data(source='astropy')`, which can be used to indicate that the only required data is from the http://data.astropy.org server. To enable just these tests, you can run the tests with `pytest --remote-data=astropy`.

For more information on the `pytest-remotedata` plugin, see pytest-remotedata.

Examples

```
from ...config import get_data_filename

def test_1():
    """Test version using a local file."""
    #if filename.fits is a local file in the source distribution
    datafile = get_data_filename('filename.fits')
    # do the test

@pytest.mark.remote_data
def test_2():
    """Test version using a remote file."""
    #this is the hash for a particular version of a file stored on
the
    #astropy data server.
    datafile =
get_data_filename('hash/94935ac31d585f68041c08f87d1a19d4')
    # do the test

def doctest_example():
    """
    >>> datafile = get_data_filename('hash/94935')  # doctest:
+REMOTE_DATA
    """
    pass
```

The `get_remote_test_data` will place the files in a temporary directory indicated by the `tempfile` module, so that the test files will eventually get removed by the system. In the long term, once test data files become too large, we will need to design a mechanism for removing test data immediately.

Tests that use the file cache

By default, the Astropy test runner sets up a clean file cache in a temporary directory that is used only for that test run and then destroyed. This is to ensure consistency between test runs, as well as to not clutter users' caches (i.e. the cache directory returned by **`get_cache_dir`**) with test files.

However, some test authors (especially for affiliated packages) may find it desirable to cache files downloaded during a test run in a more permanent location (e.g. for large data sets). To this end the **`set_temp_cache`** helper may be used. It can be used either as a context manager within a test to temporarily set the cache to a custom location, or as a *decorator* that takes effect for an entire test function (not including setup or teardown, which would have to be decorated separately).

Furthermore, it is possible to change the location of the cache directory for the duration of the test run by setting the `XDG_CACHE_HOME` environment variable.

Tests that create files

Tests may often be run from directories where users do not have write permissions so tests which create files should always do so in temporary directories. This can be done with the pytest 'tmpdir' fixture or with Python's built-in tempfile module.

Setting up/Tearing down tests

In some cases, it can be useful to run a series of tests requiring something to be set up first. There are four ways to do this:

Module-level setup/teardown

If the `setup_module` and `teardown_module` functions are specified in a file, they are called before and after all the tests in the file respectively. These functions take one argument, which is the module itself, which makes it very easy to set module-wide variables:

```
def setup_module(module):
    """Initialize the value of NUM."""
    module.NUM = 11

def add_num(x):
    """Add pre-defined NUM to the argument."""
    return x + NUM

def test_42():
    """Ensure that add_num() adds the correct NUM to its argument."""
    added = add_num(42)
    assert added == 53
```

We can use this for example to download a remote test data file and have all the functions in the file access it:

```
import os

def setup_module(module):
    """Store a copy of the remote test file."""
    module.DATAFILE =
get_remote_test_data('94935ac31d585f68041c08f87d1a19d4')

def test():
    """Perform test using cached remote input file."""
    f = open(DATAFILE, 'rb')
    # do the test

def teardown_module(module):
    """Clean up remote test file copy."""
    os.remove(DATAFILE)
```

Class-level setup/teardown

Tests can be organized into classes that have their own setup/teardown functions. In the following

```
def add_nums(x, y):
    """Add two numbers."""
    return x + y

class TestAdd42(object):
    """Test for add_nums with y=42."""

    def setup_class(self):
        self.NUM = 42

    def test_1(self):
        """Test behavior for a specific input value."""
        added = add_nums(11, self.NUM)
        assert added == 53

    def test_2(self):
        """Test behavior for another input value."""
        added = add_nums(13, self.NUM)
        assert added == 55

    def teardown_class(self):
        pass
```

In the above example, the `setup_class` method is called first, then all the tests in the class, and finally the `teardown_class` is called.

Method-level setup/teardown

There are cases where one might want setup and teardown methods to be run before and after *each* test. For this, use the `setup_method` and `teardown_method` methods:

```
def add_nums(x, y):
    """Add two numbers."""
    return x + y

class TestAdd42(object):
    """Test for add_nums with y=42."""

    def setup_method(self, method):
        self.NUM = 42

    def test_1(self):
    """Test behavior for a specific input value."""
        added = add_nums(11, self.NUM)
        assert added == 53

    def test_2(self):
    """Test behavior for another input value."""
        added = add_nums(13, self.NUM)
        assert added == 55

    def teardown_method(self, method):
        pass
```

Function-level setup/teardown

Finally, one can use `setup_function` and `teardown_function` to define a setup/teardown mechanism to be run before and after each function in a module. These take one argument, which is the function being tested:

```
def setup_function(function):
    pass

def test_1(self):
   """First test."""
    # do test

def test_2(self):
    """Second test."""
    # do test

def teardown_function(function):
    pass
```

Property-based tests

Property-based testing lets you focus on the parts of your test that matter, by making more general claims - “works for any two numbers” instead of “works for 1 + 2”. Imagine if random testing gave you minimal, non-flaky failing examples, and a clean way to describe even the most complicated data - that’s property-based testing!

`pytest-astropy` includes a dependency on Hypothesis, so installation is easy - you can just read the docs or work through the tutorial and start writing tests like:

```
from astropy.coordinates import SkyCoord
from hypothesis import given, strategies as st

@given(
    st.builds(SkyCoord, ra=st.floats(0, 360), dec=st.floats(-90, 90))
)
def test_coordinate_transform(coord):
    """Test that sky coord can be translated from ICRS to Galactic
and back."""
    assert coord == coord.galactic.icrs  # floating-point precision
alert!
```

Other properties that you could test include:

- Round-tripping from image to sky coordinates and back should be lossless for distortion-free mappings, and otherwise always below 10^-5 px.
- Take a moment in time, round-trip it through various frames, and check it hasn’t changed or lost precision. (or at least not by more than a nanosecond)
- IO routines losslessly round-trip data that they are expected to handle
- Optimised routines calculate the same result as unoptimised, within tolerances

This is a great way to start contributing to Astropy, and has already found bugs in time handling. See issue #9017 and pull request #9532 for details!

(and if you find Hypothesis useful in your research, please cite it!)

Parametrizing tests

If you want to run a test several times for slightly different values, you can use `pytest` to avoid writing separate tests. For example, instead of writing:

```
def test1():
    assert type('a') == str

def test2():
    assert type('b') == str

def test3():
    assert type('c') == str
```

You can use the `@pytest.mark.parametrize` decorator to concisely create a test function for each input:

```
@pytest.mark.parametrize(('letter'), ['a', 'b', 'c'])
def test(letter):
    """Check that the input is a string."""
    assert type(letter) == str
```

As a guideline, use `parametrize` if you can enumerate all possible test cases and each failure would be a distinct issue, and Hypothesis when there are many possible inputs or you only want a single simple failure to be reported.

Tests requiring optional dependencies

For tests that test functions or methods that require optional dependencies (e.g. Scipy), pytest should be instructed to skip the test if the dependencies are not present. The following example shows how this should be done:

```
import pytest

try:
    import scipy
    HAS_SCIPY = True
except ImportError:
    HAS_SCIPY = False

@pytest.mark.skipif('not HAS_SCIPY')
```

```
def test_that_uses_scipy():
    ...
```

In this way, the test is run if Scipy is present, and skipped if not. No tests should fail simply because an optional dependency is not present.

Using pytest helper functions

If your tests need to use pytest helper functions, such as `pytest.raises`, import `pytest` into your test module like so:

```
import pytest
```

Testing warnings

In order to test that warnings are triggered as expected in certain situations, pytest provides its own context manager pytest.warns that, completely analogously to `pytest.raises` (see below) allows to probe explicitly for specific warning classes and, through the optional `match` argument, messages. Note that when no warning of the specified type is triggered, this will make the test fail. When checking for optional, but not mandatory warnings, `pytest.warns(None)` can be used to catch and inspect them.

Note

With pytest there is also the option of using the recwarn function argument to test that warnings are triggered within the entire embedding function. This method has been found to be problematic in at least one case (pull request 1174).

Testing exceptions

Just like the handling of warnings described above, tests that are designed to trigger certain errors should verify that an exception of the expected type is raised in the expected place. This is efficiently done by running the tested code inside the pytest.raises context manager. Its optional `match` argument allows to check the error message for any patterns using `regex` syntax. For example the matches `pytest.raises(OSError, match=r'^No such file')` and `pytest.raises(OSError, match=r'or directory$')` would be equivalent to `assert str(err).startswith(No such file)` and `assert str(err).endswith(or directory)`, respectively, on the raised error message `err`. For matching multi-line messages you need to pass the `(?s)` flag to the underlying `re.search`, as in the example below:

```
with pytest.raises(fits.VerifyError, match=r'(?s)not upper.+ Illegal
key') as excinfo:
    hdu.verify('fix+exception')
assert str(excinfo.value).count('Card') == 2
```

This invocation also illustrates how to get an `ExceptionInfo` object returned to perform additional diagnostics on the info.

Testing configuration parameters

In order to ensure reproducibility of tests, all configuration items are reset to their default values when the test runner starts up.

Sometimes you'll want to test the behavior of code when a certain configuration item is set to a particular value. In that case, you can use the **`astropy.config.ConfigItem.set_temp`** context manager to temporarily set a configuration item to that value, test within that context, and have it automatically return to its original value.

For example:

```
def test_pprint():
    from ... import conf
    with conf.set_temp('max_lines', 6):
        # ...
```

Marking blocks of code to exclude from coverage

Blocks of code may be ignored by the coverage testing by adding a comment containing the phrase `pragma: no cover` to the start of the block:

```
if this_rarely_happens:  # pragma: no cover
    this_call_is_ignored()
```

Image tests with pytest-mpl

Running image tests

We make use of the pytest-mpl plugin to write tests where we can compare the output of plotting commands with reference files on a pixel-by-pixel basis (this is used for instance in astropy.visualization.wcsaxes).

To run the Astropy tests with the image comparison, use:

```
pytest --mpl --remote-data
```

However, note that the output can be very sensitive to the version of Matplotlib as well as all its dependencies (e.g. freetype), so we recommend running the image tests inside a Docker container which has a frozen set of package versions (Docker containers can be thought of as mini virtual machines). We have made a set of Docker container images that can be used for this. Once you have installed Docker, to run the Astropy tests with the image comparison inside a Docker container, make sure you are inside the Astropy repository (or the repository of the package you are testing) then do:

```
docker run -it -v ${PWD}:/repo astropy/image-tests-py35-mpl300:1.3
/bin/bash
```

This will start up a bash prompt in the Docker container, and you should see something like:

```
root@8173d2494b0b:/#
```

You can now go to the `/repo` directory, which is the same folder as your local version of the repository you are testing:

```
cd /repo
```

You can then run the tests as above:

```
pip install -e .[test]
pytest --mpl --remote-data
```

Type `exit` to exit the container.

You can find the names of the available Docker images on the Docker Hub.

Writing image tests

The README.rst for the plugin contains information on writing tests with this plugin. The only key addition compared to those instructions is that you should set `baseline_dir`:

```
from astropy.tests.image_tests import IMAGE_REFERENCE_DIR

@pytest.mark.mpl_image_compare(baseline_dir=IMAGE_REFERENCE_DIR)
```

This is because since the reference image files would contribute significantly to the repository size, we instead store them on the http://data.astropy.org site. The downside is that it is a little more complicated to create or re-generate reference files, but we describe the process here.

Generating reference images

Once you have a test for which you want to (re-)generate reference images, start up one of the Docker containers using e.g.:

```
docker run -it -v ${PWD}:/repo astropy/image-tests-py35-mpl300:1.3
/bin/bash
```

then run the tests inside `/repo` with the `--mpl-generate-path` argument, e.g:

```
cd repo
pip install -e .[test]
pytest --mpl --mpl-generate-path=reference_tmp --remote-data
```

This will create a `reference_tmp` folder and put the generated reference images inside it - the folder will be available in the repository outside of the Docker container. Type `exit` to exit the container.

Make sure you generate images for the different supported Matplotlib versions using the available containers.

Uploading the reference images

Next, we need to add these images to the http://data.astropy.org server. To do this, open a pull request to this repository. The reference images for Astropy tests should go inside the testing/astropy directory. In that directory are folders named as timestamps. If you are simply adding new tests, add the reference files to the most recent directory.

If you are re-generating baseline images due to changes in Astropy, make a new timestamp directory by copying one the most recent one, then replace any baseline images that have changed. Note that due to changes between Matplotlib versions, we need to add the whole set of reference images for each major Matplotlib version. Therefore, in each timestamp folder, there are folders named e.g. `1.4.x` and `1.5.x`.

Once the reference images are merged in and available on http://data.astropy.org, update the timestamp in the `IMAGE_REFERENCE_DIR` variable in the `astropy.tests.image_tests` sub-module. Because the timestamp is hard-coded, adding a new timestamp directory will not mess with testing for released versions of Astropy, so you can easily add and tweak a new timestamp directory while still working on a pull request to Astropy.

Writing doctests

A doctest in Python is a special kind of test that is embedded in a function, class, or module’s docstring, or in the narrative Sphinx documentation, and is formatted to look like a Python interactive session–that is, they show lines of

Python code entered at a `>>>` prompt followed by the output that would be expected (if any) when running that code in an interactive session.

The idea is to write usage examples in docstrings that users can enter verbatim and check their output against the expected output to confirm that they are using the interface properly.

Furthermore, Python includes a **`doctest`** module that can detect these doctests and execute them as part of a project's automated test suite. This way we can automatically ensure that all doctest-like examples in our docstrings are correct.

The Astropy test suite automatically detects and runs any doctests in the astropy source code or documentation, or in packages using the Astropy test running framework. For example doctests and detailed documentation on how to write them, see the full **`doctest`** documentation.

Note

Since the narrative Sphinx documentation is not installed alongside the astropy source code, it can only be tested by running `pytest` directly (or via tox), not by `import astropy; astropy.test()`.

For more information on the `pytest-doctestplus` plugin used by Astropy, see pytest-doctestplus.

Skipping doctests

Sometimes it is necessary to write examples that look like doctests but that are not actually executable verbatim. An example may depend on some external conditions being fulfilled, for example. In these cases there are a few ways to skip a doctest:

1. Next to the example add a comment like: `# doctest: +SKIP`. For example:

```
>>> import os
>>> os.listdir('.')  # doctest: +SKIP
```

   In the above example we want to direct the user to run `os.listdir('.')` but we don't want that line to be executed as part of the doctest.

   To skip tests that require fetching remote data, use the `REMOTE_DATA` flag instead. This way they can be turned on using the `--remote-data` flag when running the tests:

```
>>> datafile = get_data_filename('hash/94935')  # doctest:
+REMOTE_DATA
```

2. Astropy's test framework adds support for a special `__doctest_skip__` variable that can be placed at the module level of any module to list functions, classes, and methods in that module whose doctests should not be run. That is, if it doesn't make sense to run a function's example usage as a doctest, the entire function can be skipped in the doctest collection phase.

   The value of `__doctest_skip__` should be a list of wildcard patterns for all functions/classes whose doctests should be skipped. For example:

   ```
   __doctest_skip__ = ['myfunction', 'MyClass', 'MyClass.*']
   ```

   skips the doctests in a function called `myfunction`, the doctest for a class called `MyClass`, and all *methods* of `MyClass`.

   Module docstrings may contain doctests as well. To skip the module-level doctests include the string `'.'` in `__doctest_skip__`.

   To skip all doctests in a module:

   ```
   __doctest_skip__ = ['*']
   ```

3. In the Sphinx documentation, a doctest section can be skipped by making it part of a `doctest-skip` directive:

   ```
   .. doctest-skip::

       >>> # This is a doctest that will appear in the documentation,
       >>> # but will not be executed by the testing framework.
       >>> 1 / 0  # Divide by zero, ouch!
   ```

   It is also possible to skip all doctests below a certain line using a `doctest-skip-all` comment. Note the lack of `::` at the end of the line here:

   ```
   .. doctest-skip-all

   All doctests below here are skipped...
   ```

4. `__doctest_requires__` is a way to list dependencies for specific doctests. It should be a dictionary mapping wildcard patterns (in the same format as `__doctest_skip__`) to a list of one or more modules that should be *importable* in order for the tests to run. For example, if some tests

require the scipy module to work they will be skipped unless `import scipy` is possible. It is also possible to use a tuple of wildcard patterns as a key in this dict:

```
__doctest_requires__ = {('func1', 'func2'): ['scipy']}
```

Having this module-level variable will require `scipy` to be importable in order to run the doctests for functions `func1` and `func2` in that module.

In the Sphinx documentation, a doctest requirement can be notated with the `doctest-requires` directive:

```
.. doctest-requires:: scipy

    >>> import scipy
    >>> scipy.hamming(...)
```

Skipping output

One of the important aspects of writing doctests is that the example output can be accurately compared to the actual output produced when running the test.

The doctest system compares the actual output to the example output verbatim by default, but this not always feasible. For example the example output may contain the `__repr__` of an object which displays its id (which will change on each run), or a test that expects an exception may output a traceback.

The simplest way to generalize the example output is to use the ellipses `...`. For example:

```
>>> 1 / 0
Traceback (most recent call last):
...
ZeroDivisionError: integer division or modulo by zero
```

This doctest expects an exception with a traceback, but the text of the traceback is skipped in the example output–only the first and last lines of the output are checked. See the **doctest** documentation for more examples of skipping output.

Ignoring all output

Another possibility for ignoring output is to use the `# doctest: +IGNORE_OUTPUT` flag. This allows a doctest to execute (and check that the code executes without errors), but allows the entire output to be ignored in cases where we don't care what the output is. This differs from using ellipses in that we can still provide complete example output, just without the test checking that it is exactly right. For example:

```
>>> print('Hello world')
We don't really care what the output is as long as there were no
errors...
```

Handling float output

Some doctests may produce output that contains string representations of floating point values. Floating point representations are often not exact and contain roundoffs in their least significant digits. Depending on the platform the tests are being run on (different Python versions, different OS, etc.) the exact number of digits shown can differ. Because doctests work by comparing strings this can cause such tests to fail.

To address this issue, the `pytest-doctestplus` plugin provides support for a `FLOAT_CMP` flag that can be used with doctests. For example:

```
>>> 1.0 / 3.0  # doctest: +FLOAT_CMP
0.333333333333333311
```

When this flag is used, the expected and actual outputs are both parsed to find any floating point values in the strings. Those are then converted to actual Python **`float`** objects and compared numerically. This means that small differences in representation of roundoff digits will be ignored by the doctest. The values are otherwise compared exactly, so more significant (albeit possibly small) differences will still be caught by these tests.

Continuous integration

Overview

Astropy uses the following continuous integration (CI) services:

- Travis for 64-bit Linux, OS X, and Windows setups
- CircleCI for 32-bit Linux, documentation build, and visualization tests

These continuously test the package for each commit and pull request that is pushed to GitHub to notice when something breaks.

Astropy and many affiliated packages use an external package called ci-helpers to provide support for the generic parts of the CI systems. `ci-helpers` consists of a set of scripts that are used by the `.travis.yml` and `appveyor.yml` files to set up the conda environment, and install dependencies.

Dependencies can be customized for different packages using the appropriate environment variables in `.travis.yml` and `appveyor.yml`. For more details on how to set up this machinery, see the package-template and ci-helpers.

The 32-bit tests on CircleCI use the quay.io/pypa/manylinux1_i686 docker image which includes a 32-bit Python environment for each major Python version. See the CircleCI configuration file for the core package for how to access the different Python versions.

In some cases, you may see failures on continuous integration services that you do not see locally, for example because the operating system is different, or because the failure happens with only 32-bit Python. The following sections explain how you can reproduce specific builds locally.

Reproducing failing 32-bit builds

If you want to run your tests in the same 32-bit Python environment that CircleCI uses, start off by installing Docker if you don't already have it installed. Docker can be installed on a variety of different operating systems.

Then, make sure you have a version of the git repository (either the main Astropy repository or your fork) for which you want to run the tests. Go to that directory, then run Docker with:

```
$ docker run -i -v ${PWD}:/astropy_src -t quay.io/pypa
/manylinux1_i686 bash
```

This will put you in the bash shell inside the Docker container. Once inside, you can go to the `/astropy_src` directory, and you should see the files that are in your local git repository:

```
root@5e2b89d7b07c:/# cd /astropy_src
root@5e2b89d7b07c:/astropy_src# ls
astropy                    conftest.py      licenses
setup.py
cextern                    CONTRIBUTING.md  MANIFEST.in       static
CHANGES.rst          docs             pip-requirements  tox.ini
CITATION             examples             pyproject.toml
codecov.yml          GOVERNANCE.md    README.rst
CODE_OF_CONDUCT.md  LICENSE.rst      setup.cfg
```

You can then run the tests with e.g.:

```
root@5e2b89d7b07c:/astropy_src# /opt/python/cp36-cp36m/bin/pip
install -e .[test]
root@5e2b89d7b07c:/astropy_src# pytest
```

Pytest Plugins

The following `pytest` plugins are maintained and used by Astropy. They are included as dependencies to the `pytest-astropy` package, which is now required for testing Astropy. More information on all of the plugins provided by the `pytest-astropy` package (including dependencies not maintained by Astropy) can be found here.

pytest-remotedata

The pytest-remotedata plugin allows developers to control whether to run tests that access data from the internet. The plugin provides two decorators that can be used to mark individual test functions or entire test classes:

- `@pytest.mark.remote_data` for tests that require data from the internet
- `@pytest.mark.internet_off` for tests that should run only when there is no internet access. This is useful for testing local data caches or fallbacks for when no network access is available.

The plugin also adds the `--remote-data` option to the `pytest` command (which is also made available through the Astropy test runner).

If the `--remote-data` option is not provided when running the test suite, or if `--remote-data=none` is provided, all tests that are marked with `remote_data` will be skipped. All tests that are marked with `internet_off` will be executed. Any test that attempts to access the internet but is not marked with `remote_data` will result in a failure.

Providing either the `--remote-data` option, or `--remote-data=any`, will cause all tests marked with `remote_data` to be executed. Any tests that are marked with `internet_off` will be skipped.

Running the tests with `--remote-data=astropy` will cause only tests that receive remote data from Astropy data sources to be run. Tests with any other data sources will be skipped. This is indicated in the test code by marking test functions with `@pytest.mark.remote_data(source='astropy')`. Tests marked with `internet_off` will also be skipped in this case.

Also see Working with data files.

pytest-doctestplus

The pytest-doctestplus plugin provides advanced doctest features, including:

- handling doctests that use remote data in conjunction with the `pytest-remotedata` plugin above (see Working with data files)
- approximate floating point comparison for doctests that produce floating

point results (see Handling float output)
- skipping particular classes, methods, and functions when running doctests (see Skipping doctests)
- optional inclusion of `*.rst` files for doctests

This plugin provides two command line options: `--doctest-plus` for enabling the advanced features mentioned above, and `--doctest-rst` for including `*.rst` files in doctest collection.

The Astropy test runner enables both of these options by default. When running the test suite directly from `pytest` (instead of through the Astropy test runner), it is necessary to explicitly provide these options when they are needed.

pytest-openfiles

The pytest-openfiles plugin allows for the detection of open I/O resources at the end of unit tests. This plugin adds the `--open-files` option to the `pytest` command (which is also exposed through the Astropy test runner).

When running tests with `--open-files`, if a file is opened during the course of a unit test but that file not closed before the test finishes, the test will fail. This is particularly useful for testing code that manipulates file handles or other I/O resources. It allows developers to ensure that this kind of code properly cleans up I/O resources when they are no longer needed.

Also see Testing for open files.

Writing Command-Line Scripts

Command-line scripts in Astropy should follow a consistent scheme to promote readability and compatibility.

Setuptools' "entry points" are used to automatically generate wrappers with the correct extension. The scripts can live in their own module, or be part of a larger module that implements a class or function for astropy library use. They should have a `main` function to parse the arguments and pass those arguments on to some library function so that the library function can be used programmatically when needed. The `main` function should accept an optional single argument that holds the `sys.argv` list, except for the script name (e.g., `argv[1:]`). It must then be added to the list of entry points in the `setup.py` file (see the example below).

Command-line options can be parsed however desired, but the **`argparse`** module is recommended when possible, due to its simpler and more flexible interface relative to the older **`optparse`**.

Example

Contents of `/astropy/somepackage/somemod.py`

```
def do_something(args, option=False):
    for a in args:
        if option:
            ...do something...
        else:
            ...do something else...

def main(args=None):

    import argparse

    parser = argparse.ArgumentParser(description='Process some
integers.')
    parser.add_argument('-o', '--option',
dest='op',action='store_true',
                        help='Some option that turns something on.')
    parser.add_argument('stuff', metavar='S', nargs='+',
                        help='Some input I should be able to get lots
of.')

    res = parser.parse_args(args)

    do_something(res.stuff,res.op)
```

Then add the script to the `setup.cfg` under this section:

```
[options.entry_points]
console_scripts =
    somescript = astropy.somepackage.somemod:main
```

Building Astropy and its Subpackages

The build process currently uses the setuptools package to build and install the astropy core (and any affiliated packages that use the template). As is typical, there is a single `setup.py` file that is used for the whole `astropy` package. To make it easier to set up C extensions for individual sub-packages, we use extension-helpers, which allows extensions to be defined inside each sub-package.

The way extension-helpers works is that it looks for `setup_package.py` files anywhere in the package, and then looks for a function called `get_extensions` inside each of these files. This function should return a list of `setuptools.Extension` objects, and these are combined into an overall list of extensions to build.

For certain string-parsing tasks, Astropy uses the PLY tool. PLY generates tables that speed up the parsing process, which are checked into source code so they don't have to be regenerated. These tables can be recognized by having either `lextab` or `parsetab` in their names. To regenerate these files (e.g. if a new version of PLY is bundled with Astropy or some of the parsing code changes), the tables need to be deleted and the appropriate parts of astropy re-imported and run. For exact details, see the comments in the headers of the `parsetab` and `lextab` files.

Building on Windows

The most convenient option is to use Python installation from Miniconda. If you like Unix-like commands, Git Bash, which comes installed with Git, complements Miniconda pretty well, as long as Miniconda is installed with the option for it to be available system-wide (the option that is not recommended by the installer).

Since `astropy` contains C extensions, you also need to install Microsoft Visual Studio (the latest available should work) so Python can access the system C compiler.

Once everything is set up as above, you can proceed to build `astropy` from source in the `conda` environment in an OS-agnostic way. For example:

- Create a new `conda` environment.
- Go to the `astropy` code checkout directory.
- If you have not already, fetch all of the tags from the main repository. If you do not have the latest tag, your developer version number will be wrong.
- Run `pip install -e .` to build `astropy`.

C or Cython Extensions

Astropy supports using C extensions for wrapping C libraries and Cython for speeding up computationally-intensive calculations. Both Cython and C extension building can be customized using the `get_extensions` function of the `setup_package.py` file. If defined, this function must return a list of `setuptools.Extension` objects. The creation process is left to the subpackage designer, and can be customized however is relevant for the extensions in the subpackage.

While C extensions must always be defined through the `get_extensions` mechanism, Cython files (ending in `.pyx`) are automatically located by extension-helpers and loaded in separate extensions if they are not in `get_extensions`. For Cython extensions located in this way, headers for numpy C functions are included in the build, but no other external headers are included. `.pyx` files present in the extensions returned by `get_extensions` are not included in the list of automatically generated extensions.

Note

If a `setuptools.Extension` object is provided for Cython source files using the `get_extensions` mechanism, it is very important that the `.pyx` files be given as the `source`, rather than the `.c` files generated by Cython.

Using Numpy C headers

If your C or Cython extensions uses **numpy** at the C level, you probably need access to the numpy C headers. When doing this, you should use `numpy.get_include()` to specify the include directory to use, for example:

```
from setuptools import Extension
import numpy

def get_extensions():
    return Extension(name='myextension', sources=['myext.c'],
                     include_dirs=[numpy.get_include()])
```

Installing C header files

If your C extension needs to be linked from other third-party C code, you probably want to install its header files along side the Python module.

1. Create an `include` directory inside of your package for all of the header files.
2. Use the `[options.package_data]` section in your `setup.cfg` file to include those header files in the package. For example, the **astropy.wcs** package has the following entries in the `[options.package_data]` section:

```
[options.package_data]
...
astropy.wcs = include/*/*.h
...
```

Preventing importing at build time

It is important to make sure that `setup_package.py` files do not trigger an import of the package they are in - so they should be able to be executed without relying on imports to other parts of the package.

Speed up your builds with ccache

ccache is a tool that caches compiled sources so that they don't have to be recompiled (so long as they are unchanged) even if the outputs have been deleted. This means that if you switch branches or clean your source checkout you can save a lot of time by avoiding the majority of re-compiles from scratch.

Because installation and configuration of ccache varies from platform to platform, please consult the ccache documentation and/or Google to set up ccache on your system–this is strongly encouraged for anyone doing significant development of Astropy or scientific programming in general.

Release Procedures

The current release procedure for Astropy involves a combination of an automated release script and some manual steps. Future versions will automate more of the process, if not all.

There are several different procedures below, depending on the situation:

For a signed release, see Creating a GPG Signing Key and a Signed Tag for relevant setup instructions.

Standard Release Procedure

This is the standard release procedure for releasing Astropy (or affiliated packages that use the full bugfix/maintenance branch approach.)

1. Create a GitHub milestone for the next bugfix version, move any remaining issues from the version you are about to release, and close the milestone. When releasing a major release, close the last milestone on the previous maintenance branch, too.

   Note

Creation of new milestone can be done as early as when you ping maintainers about their relevant pull requests, so that the maintainers have the option to re-milestone their work.

2. If there are any issues in the GitHub issue tracker that are labeled `affects-dev` but are issues that apply to this release, update them to `affects-release`. Similarly, if any issues remain open for this release, re-assign them to the next relevant milestone.
3. (Only for major versions) Make sure to update the "What's new" section with the stats on the number of issues, PRs, and contributors. For the first two, the astropy-procedures repository script `gh_issuereport.py` can provide the numbers since the last major release. For the final one, you will likely need to update the Astropy `.mailmap` file, as there are often contributors who are not careful about using the same e-mail address for every commit. The easiest way to do this is to run the command `git shortlog -n -s -e` to see the list of all contributors and their email addresses. Look for any misnamed entries or duplicates, and add them to the `.mailmap` file (matched to the appropriate canonical name/email address.) Once you have finished this, you can count the number of lines in `git shortlog -s` to get the final contributor count.
4. Also be sure to update the `docs/credits.rst` file to include any new contributors from the above step. (This step is only required on major releases, but can be done for bugfix releases as time allows.)
5. (astropy specific) Ensure the built-in IERS earth rotation parameter and leap second tables are up to date by changing directory to `astropy/utils/iers/data` and executing `update_builtin_iers.sh`. Check the result with `git diff` (do not be surprised to find many lines in the `eopc04_IAU2000.62-now` file change; those data are reanalyzed periodically) and committing.
6. Ensure you have a GPG key pair available for when git needs to sign the tag you create for the release. See Creating a GPG Signing Key and a Signed Tag for more on this.
7. Obtain a *clean* version of the astropy core repository. That is, one where you don't have any intermediate build files. Either use a fresh `git clone` or do `git clean -dfx`.
8. Be sure you're on the branch appropriate for the version you're about to release. For example, if releasing version 1.2.2 make sure to:

```
$ git checkout v1.2.x
```

9. Make sure that the continuous integration services (e.g., Travis or CircleCI) are passing for the astropy core repository branch you are going to release. Also check that the Azure core package pipeline which builds wheels on the `v*` branches is passing. You may also want to locally run the tests (with remote data on to ensure all of the tests actually run), using tox to do a thorough test in an isolated environment:

```
$ pip install tox --upgrade
$ TEST_READ_HUGE_FILE=1 tox -e test-alldeps -- --remote-data=any
```

10. Edit the `CHANGES.rst` file by changing the date for the version you are about to release from “unreleased” to today’s date. Also be sure to remove any sections of the changelog for that version that have no entries. For releases that come after release candidates (Modifications for a beta/release candidate release), the title of the changelog section should be replaced too, thus getting rid of any mention of the release candidate. Then add and commit those changes with:

```
<use your favorite editor on CHANGES.rst>
$ git add CHANGES.rst
$ git commit -m "Finalizing changelog for v<version>"
```

11. Push the branch back to GitHub, e.g.:

```
$ git push upstream v1.2.x
```

and make sure that the CI services mentioned above (includnig the Azure pipeline) are still passing.

Note

You may need to replace `upstream` here with `astropy` or whatever remote name you use for the astropy core repository.

12. Tag the commit with `v<version>`, being certain to sign the tag with the `-s` option:

```
$ git tag -s v<version> -m "Tagging v<version>"
```

13. Push up the tag to the astropy core repository:

```
$ git push upstream v<tag version>
```

Note

You may need to replace `upstream` here with `astropy` or whatever remote name you use for the astropy core repository. Also, it might be tempting to use the `--tags` argument to `git push`, but this should *not* be done, as it might push up some unintended tags.

At this point if all goes well, the wheels and sdist will be build in the Azure core package pipeline and uploaded to PyPI!

14. In the event there are any issues with the wheel building for the tag (which shouldn't really happen if it was passing for the release branch), you'll have to fix whatever the problem is. First you will need to back out the release procedure by dropping the commits you made for release and removing the tag you created:

```
$ git reset --hard HEAD^^^^ # you could also use the SHA hash of
the commit before your first changelog edit
$ git tag -d v<version>
```

Note

Any re-pushing the same tag back out to GitHub hereafter would be a force-push.

Once the sdist and wheels are uploaded, the release is done!

Congratulations! You have completed the release! Now there are just a few clean-up tasks to finalize the process.

Post-Release procedures

1. Go back to release branch (e.g., `1.2.x`) and update the `CHANGES.rst` file with a new section for the next version. Then add and commit:

```
$ git checkout v1.2.x
<use your favorite editor on CHANGES.rst>
$ git add CHANGES.rst
$ git commit -m "Add v<next_version> to the changelog"
```

2. Push up these changes to the astropy core repository:

```
$ git push upstream v<version branch>.x
```

3. If this is a release of the current release (i.e., not an LTS supported along side a more recent version), update the "stable" branch to point to the new release:

```
$ git checkout stable
$ git reset --hard v<version>
$ git push upstream stable --force
```

4. Update Readthedocs so that it builds docs for the version you just released. You'll find this in the "admin" tab, with checkboxes next to each github tag. Also verify that the `stable` Readthedocs version builds correctly for the new version (it should trigger automatically once you've done the previous step).
5. When releasing a patch release, also set the previous RTD version in the release history to "protected". For example when releasing v1.1.2, set v1.1.1 to "protected". This prevents the previous releases from cluttering the list of versions that users see in the version dropdown (the previous versions are still accessible by their URL though).
6. Update the Astropy web site by editing the `index.html` page at https://github.com/astropy/astropy.github.com by changing the "current version" link and/or updating the list of older versions if this is an LTS bugfix or a new major version. You may also need to update the contributor list on the web site if you updated the `docs/credits.rst` at the outset.
7. Open a PR to the astropy *master* branch to update the `CHANGES.rst` to reflect the date of the release you just performed and to include the new section of the changelog. Often the easiest way to do this is to use `git cherry-pick` the changelog commit just before the release commit from above. If you are not sure how to do this, you might be better off copying-and-pasting the relevant parts of the maintenance branch's `CHANGES.rst` into master. In the same PR, you also have to update `docs/whatsnew/index.rst` and `docs/whatsnew/X.Y.rst` to link to "what's new" documentation in the released RTD branch, using the existing text as example.
8. `conda-forge` has a bot that automatically opens a PR from a new PyPI (stable) release, which you need to follow up on and merge. Meanwhile, for a LTS release, you still have to manually open a PR at astropy-feedstock. This is similar to the process for wheels. When the `conda-forge` package is ready, email the Anaconda maintainers about the release(s) so they can update the versions in the default channels. Typically, you should wait to make sure `conda-forge` and possibly `conda` works before sending out the public announcement (so that users who want to try out the new version can do so with `conda`).
9. Update the `LATEST_ASTROPY_STABLE` or `ASTROPY_LTS_VERSION` variables in the `ci-helpers` repository once the `conda` packages

became available.

10. Upload the release to Zenodo. This has to be done manually since the Zenodo/GitHub integration relies on making releases on GitHub, which we don't do. So for the Astropy core package, log in to Zenodo using the Astropy team credentials, then go to the existing record. Click on **New version** - note that it's important to do this rather than upload the release as a completely new record. You should now see a pre-filled deposit form with the details from the previous release. Start off by removing the existing file under the **Files** section, then click on **Choose Files** and select the `tar.gz` release file for the core package release you are uploading, and click **Start upload**. Before you publish this, there are a few fields to update in the form: the **Publication date** should be set to the date the tar file was uploaded to PyPI, the **Title** should be updated to include the new version number, and the **Version** should be updated to include the version number (with no `v` prefix). Once you are happy with the changes, click **Save**, then **Publish**.
11. Once the release(s) are available on the default `conda` channels, prepare the public announcement. Use the previous announcement as a template, but link to the release tag instead of `stable`. For a new major release, you should coordinate with the Astropy Coordinators. Meanwhile, for a bugfix release, you can proceed to send out an email to the `astropy-dev` and Astropy mailing lists.

Modifications for a beta/release candidate release

For major releases, we do beta and/or release candidates to have a chance to catch significant bugs before the true release. If the release you are performing is this kind of pre-release, some of the above steps need to be modified.

The primary modifications to the release procedure are:

- When entering tagging the release, include a `b?` or `rc??` suffix after the version number, e.g. “1.2b1” or “1.2rc1”. It is critical that you follow this numbering scheme (`X.Yb#` or `X.Y.Zrc#`), as it will ensure the release is ordered “before” the main release by various automated tools, and also tells PyPI that this is a “pre-release.”
- Do not do steps in Post-Release procedures.

Once a release candidate is available, create a new Wiki page under Astropy Project Wiki with the title “vX.Y RC testing” (replace “X.Y” with the release number) using the wiki of a previous RC as a template.

Performing a Feature Freeze/Branching new Major Versions

As outlined in APE2, astropy releases occur at regular intervals, but feature freezes occur well before the actual release. Feature freezes are also the time when the master branch's development separates from the new major version's maintenance branch. This allows new development for the next major version to continue while the soon-to-be-released version can focus on bug fixes and documentation updates.

The procedure for this is straightforward:

1. Update your local master branch to use to the latest version from github:

   ```
   $ git fetch upstream --tags
   $ git checkout -B master upstream/master
   ```

2. Create a new branch from master at the point you want the feature freeze to occur:

   ```
   $ git branch v<version>.x
   ```

3. Update the `CHANGES.rst` file with a new section at the very top for the next major version. Then add and commit those changes:

   ```
   <use your favorite editor on CHANGES.rst>
   $ git add CHANGES.rst
   $ git commit -m "Add <next_version> to changelog"
   ```

4. Tag this commit using the next major version followed by `.dev`. For example, if you have just branched `4.0`, create the `v4.1.dev` tag on the commit adding the `4.1` section to the changelog:

   ```
   $ git tag -s "v<next_version>.dev" -m "Back to development:
   v<next_version>"
   ```

5. Also update the "what's new" section of the docs to include a section for the next major version. E.g.:

   ```
   $ cp docs/whatsnew/<current_version>.rst docs/whatsnew
   /<next_version>.rst
   ```

   You'll then need to edit `docs/whatsnew/<next_version>.rst`, removing all the content but leaving the basic structure. You may also need to replace the "by the numbers" numbers with "xxx" as a reminder to update them before the next release. Then add the new version to the top of `docs/whatsnew/index.rst`, update the reference in `docs/index.rst` to point to the that version, and commit these changes

```
$ git add docs/whatsnew/<next_version>.rst
$ git add docs/whatsnew/index.rst
$ git add docs/index.rst
$ git commit -m "Added <next_version> whats new section"
```

6. Push all of these changes up to github:

```
$ git push upstream v<version>.x:v<version>.x
$ git push upstream master:master
```

Note

You may need to replace `upstream` here with `astropy` or whatever remote name you use for the astropy core repository.

7. On the github issue tracker, add a new milestone for the next major version.

Maintaining Bug Fix Releases

Note

Always start with LTS release, followed by, if necessary, a bugfix for stable release. If the releases are not done in that order, the change log entries on what goes where can get mixed up.

Astropy releases, as recommended for most Python projects, follows a <major>.<minor>.<micro> version scheme, where the “micro” version is also known as a “bug fix” release. Bug fix releases should not change any user-visible interfaces. They should only fix bugs on the previous major/minor release and may also refactor internal APIs or include omissions from previous releases–that is, features that were documented to exist but were accidentally left out of the previous release. They may also include changes to docstrings that enhance clarity but do not describe new features (e.g., more examples, typo fixes, etc).

Bug fix releases are typically managed by maintaining one or more bug fix branches separate from the master branch (the release procedure below discusses creating these branches). Typically, whenever an issue is fixed on the Astropy master branch a decision must be made whether this is a fix that should be included in the Astropy bug fix release. Usually the answer to this question is “yes”, though there are some issues that may not apply to the bug fix branch. For example, it is not necessary to backport a fix to a new feature that did not exist when the bug fix branch was first created. New features are never merged into the bug fix branch–only bug fixes; hence the name.

In rare cases a bug fix may be made directly into the bug fix branch without going into the master branch first. This may occur if a fix is made to a feature that has been removed or rewritten in the development version and no longer has the issue being fixed. However, depending on how critical the bug is it may be worth including in a bug fix release, as some users can be slow to upgrade to new major/micro versions due to API changes.

Issues are assigned to an Astropy release by way of the Milestone feature in the GitHub issue tracker. At any given time there are at least two versions under development: The next major/minor version, and the next bug fix release. For example, at the time of writing there are two release milestones open: v1.2.2 and v0.3.0. In this case, v1.2.2 is the next bug fix release and all issues that should include fixes in that release should be assigned that milestone. Any issues that implement new features would go into the v0.3.0 milestone–this is any work that goes in the master branch that should not be backported. For a more detailed set of guidelines on using milestones, see Using Milestones and Labels.

Backporting fixes from master

Note

The changelog script in `astropy-procedures` (`pr_consistency` scripts in particular) does not know about minor releases, thus please be careful. For example, let's say we have two branches (`master` and `v1.2.x`). Both 1.2.0 and 1.2.1 releases will come out of the same v1.2.x branch. If a PR for 1.2.1 is merged into `master` before 1.2.0 is released, it should not be backported into v1.2.x branch until after 1.2.0 is released, despite complaining from the aforementioned script. This situation only arises in a very narrow time frame after 1.2.0 freeze but before its release.

Most fixes are backported using the `git cherry-pick` command, which applies the diff from a single commit like a patch. For the sake of example, say the current bug fix branch is 'v1.2.x', and that a bug was fixed in master in a commit `abcd1234`. In order to backport the fix, checkout the v1.2.x branch (it's also good to make sure it's in sync with the astropy core repository) and cherry-pick the appropriate commit:

```
$ git checkout v1.2.x
$ git pull upstream v1.2.x
$ git cherry-pick abcd1234
```

Sometimes a cherry-pick does not apply cleanly, since the bug fix branch represents a different line of development. This can be resolved like any other merge conflict: Edit the conflicted files by hand, and then run `git commit` and accept the default commit message. If the fix being cherry-picked has an associated changelog entry in a separate commit make sure to backport that as well.

What if the issue required more than one commit to fix? There are a few possibilities for this. The easiest is if the fix came in the form of a pull request that was merged into the master branch. Whenever GitHub merges a pull request it generates a merge commit in the master branch. This merge commit represents the *full* difference of all the commits in the pull request combined. What this means is that it is only necessary to cherry-pick the merge commit (this requires adding the `-m 1` option to the cherry-pick command). For example, if `5678abcd` is a merge commit:

```
$ git checkout v1.2.x
$ git pull upstream v1.2.x
$ git cherry-pick -m 1 5678abcd
```

In fact, because Astropy emphasizes a pull request-based workflow, this is the *most* common scenario for backporting bug fixes, and the one requiring the least thought. However, if you're not dealing with backporting a fix that was not brought in as a pull request, read on.

See also

Merge commits and cherry picks for further explanation of the cherry-pick command and how it works with merge commits.

If not cherry-picking a merge commit there are still other options for dealing with multiple commits. The simplest, though potentially tedious, is to run the cherry-pick command once for each commit in the correct order. However, as of Git 1.7.2 it is possible to merge a range of commits like so:

```
$ git cherry-pick 1234abcd..56789def
```

This works fine so long as the commits you want to pick are actually congruous with each other. In most cases this will be the case, though some bug fixes will involve followup commits that need to back backported as well. Most bug fixes will have an issues associated with it in the issue tracker, so make sure to reference all commits related to that issue in the commit message. That way it's harder for commits that need to be backported from getting lost.

Making fixes directly to the bug fix branch

As mentioned earlier in this section, in some cases a fix only applies to a bug fix release, and is not applicable in the mainline development. In this case there are two choices:

1. An Astropy developer with commit access to the astropy core repository may check out the bug fix branch and commit and push your fix directly.
2. **Preferable**: You may also make a pull request through GitHub against the bug fix branch rather than against master. Normally when making a pull request from a branch on your fork to the astropy core repository, GitHub compares your branch to Astropy's master. If you look on the left-hand side of the pull request page, under "base repo: astropy/astropy" there is a drop-down list labeled "base branch: master". You can click on this drop-down and instead select the bug fix branch ("v1.2.x" for example). Then GitHub will instead compare your fix against that branch, and merge into that branch when the PR is accepted.

Preparing the bug fix branch for release

There are two primary steps that need to be taken before creating a bug fix release. The rest of the procedure is the same as any other release as described in Standard Release Procedure (although be sure to provide the right version number).

1. Any existing fixes to the issues assigned to a release milestone (and older LTS releases, if there are any), must be included in the maintenance branch before release.
2. The Astropy changelog must be updated to list all issues–especially user-visible issues–fixed for the current release. The changelog should be updated in the master branch, and then merged into the bug fix branch. Most issues *should* already have changelog entries for them. But occasionally these are forgotten, so if doesn't exist yet please add one in the process of backporting. See Updating and Maintaining the Changelog for more details.

To aid this process, there are a series of related scripts in the astropy-procedures repository, in the `pr_consistency` directory. These scripts essentially check that the above two conditions are met. Detailed documentation for these scripts is given in their repository, but here we summarize the basic workflow. Run the scripts in order (they are numbered `1.<something>.py`, `2.<something>.py`, etc.), entering your github login credentials as needed (if you are going to run them multiple times, using a `~/.netrc` file is recommended - see this Stack Overflow post for more on how to do that, or a similar github help page). The script to actually check consistency should be run like:

```
$ python 4.check_consistency.py > consistency.html
```

Which will generate a simple web page that shows all of the areas where either a pull request was merged into master but is *not* in the relevant release that it has been milestoned for, as well as any changelog irregularities (i.e., PRs that are in the wrong section for what the github milestone indicates). You'll want to correct those irregularities *first* before starting the backport process (re-running the scripts in order as needed).

The end of the `consistency.html` page will then show a series of `git cherry-pick` commands to update the maintenance branch with the PRs that are needed to make the milestones and branches consistent. Make sure you're in the correct maintenance branch with e.g.,

```
$ git checkout v1.3.x
$ git pull upstream v1.3.x  # Or possibly a rebase if conflicts exist
```

if you are doing bugfixes for the 1.3.x series. Go through the commands one at a time, following the cherry-picking procedure described above. If for some reason you determine the github milestone was in error and the backporting is impossible, re-label the issue on github and move on. Also, whenever you backport a PR, it's useful to leave a comment in the issue along the lines of "backported this to v1.3.x as <SHA>" so that it's clear that the backport happened to others who might later look.

Warning

Automated scripts are never perfect, and can either miss issues that need to be backported, or in some cases can report false positives.

It's always a good idea before finalizing a bug fix release to look on GitHub through the list of closed issues in the release milestone and check that each one has a fix in the bug fix branch. Usually a quick way to do this is for each issue to run:

```
$ git log --oneline <bugfix-branch> | grep #<issue>
```

Most fixes will mention their related issue in the commit message, so this tends to be pretty reliable. Some issues won't show up in the commit log, however, as their fix is in a separate pull request. Usually GitHub makes this clear by cross-referencing the issue with its PR.

Finally, not all issues assigned to a release milestone need to be fixed before making that release. Usually, in the interest of getting a release with existing fixes out within some schedule, it's best to triage issues that won't be fixed soon to a new release milestone. If the upcoming bug fix release is 'v1.2.2', then go ahead and create a 'v1.2.3' milestone and reassign to it any issues that you don't expect to be fixed in time for 'v1.2.2'.

Creating a GPG Signing Key and a Signed Tag

One of the main steps in performing a release is to create a tag in the git repository representing the exact state of the repository that represents the version being released. For Astropy we will always use signed tags: A signed tag is annotated with the name and e-mail address of the signer, a date and time, and a checksum of the code in the tag. This information is then signed with a GPG private key and stored in the repository.

Using a signed tag ensures the integrity of the contents of that tag for the future. On a distributed VCS like git, anyone can create a tag of Astropy called “0.1” in their repository–and where it’s easy to monkey around even after the tag has been created. But only one “0.1” will be signed by one of the Astropy Project coordinators and will be verifiable with their public key.

Generating a public/private key pair

Git uses GPG to created signed tags, so in order to perform an Astropy release you will need GPG installed and will have to generated a signing key pair. Most *NIX installations come with GPG installed by default (as it is used to verify the integrity of system packages). If you don’t have the `gpg` command, consult the documentation for your system on how to install it.

For OSX, GPG can be installed from MacPorts using `sudo port install gnupg`.

To create a new public/private key pair, run:

```
$ gpg --gen-key
```

This will take you through a few interactive steps. For the encryption and expiry settings, it should be safe to use the default settings (I use a key size of 4096 just because what does a couple extra kilobytes hurt?) Enter your full name, preferably including your middle name or middle initial, and an e-mail address that you expect to be active for a decent amount of time. Note that this name and e-mail address must match the info you provide as your git configuration, so you should either choose the same name/e-mail address when you create your key, or update your git configuration to match the key info. Finally, choose a very good pass phrase that won’t be easily subject to brute force attacks.

If you expect to use the same key for some time, it’s good to make a backup of both your public and private key:

```
$ gpg --export --armor > public.key
$ gpg --export-secret-key --armor > private.key
```

Back up these files to a trusted location–preferably a write-once physical medium that can be stored safely somewhere. One may also back up their keys to a trusted online encrypted storage, though some might not find that secure enough–it's up to you and what you're comfortable with.

Add your public key to a keyserver

Now that you have a public key, you can publish this anywhere you like–in your e-mail, in a public code repository, etc. You can also upload it to a dedicated public OpenPGP keyserver. This will store the public key indefinitely (until you manually revoke it), and will be automatically synced with other keyservers around the world. That makes it easy to retrieve your public key using the gpg command-line tool.

To do this you will need your public key's keyname. To find this enter:

```
$ gpg --list-keys
```

This will output something like:

```
/path/to/.gnupg/pubring.gpg
---------------------------------------------
pub   4096D/1234ABCD 2012-01-01
uid                  Your Name <your_email>
sub   4096g/567890EF 2012-01-01
```

The 8 digit hex number on the line starting with "pub"–in this example the "1234ABCD" unique keyname for your public key. To push it to a keyserver enter:

```
$ gpg --send-keys 1234ABCD
```

But replace the 1234ABCD with the keyname for your public key. Most systems come configured with a sensible default keyserver, so you shouldn't have to specify any more than that.

Create a tag

Now test creating a signed tag in git. It's safe to experiment with this–you can always delete the tag before pushing it to a remote repository:

```
$ git tag -s v0.1 -m "Astropy version 0.1"
```

This will ask for the password to unlock your private key in order to sign the tag with it. Confirm that the default signing key selected by git is the correct one (it will be if you only have one key).

Once the tag has been created, you can verify it with:

```
$ git tag -v v0.1
```

This should output something like:

```
object e8e3e3edc82b02f2088f4e974dbd2fe820c0d934
type commit
tag v0.1
tagger Your Name <your_email> 1339779534 -0400

Astropy version 0.1
gpg: Signature made Fri 15 Jun 2012 12:59:04 PM EDT using DSA key ID
0123ABCD
gpg: Good signature from "Your Name <your_email>"
```

You can use this to verify signed tags from any repository as long as you have the signer’s public key in your keyring. In this case you signed the tag yourself, so you already have your public key.

Note that if you are planning to do a release following the steps below, you will want to delete the tag you just created, because the release script does that for you. You can delete this tag by doing:

```
$ git tag -d v0.1
```

Workflow for Maintainers

This page is for maintainers — those of us who merge our own or other peoples’ changes into the upstream repository.

Being as how you’re a maintainer, you are completely on top of the basic stuff in How to make a code contribution.

Integrating changes via the web interface (recommended)

Whenever possible, merge pull requests automatically via the pull request manager on GitHub. Merging should only be done manually if there is a really good reason to do this!

Make sure that pull requests do not contain a messy history with merges, etc. If this is the case, then follow the manual instructions, and make sure the fork is rebased to tidy the history before committing.

Integrating changes manually

First, check out the `astropy` repository. The instructions in Tell git where to look for changes in the development version of Astropy add a remote that has read-only access to the upstream repo. Being a maintainer, you've got read-write access.

It's good to have your upstream remote have a scary name, to remind you that it's a read-write remote:

```
git remote add upstream-rw git@github.com:astropy/astropy.git
git fetch upstream-rw --tags
```

Let's say you have some changes that need to go into trunk (`upstream-rw/master`).

The changes are in some branch that you are currently on. For example, you are looking at someone's changes like this:

```
git remote add someone git://github.com/someone/astropy.git
git fetch someone
git branch cool-feature --track someone/cool-feature
git checkout cool-feature
```

So now you are on the branch with the changes to be incorporated upstream. The rest of this section assumes you are on this branch.

A few commits

If there are only a few commits, consider rebasing to upstream:

```
# Fetch upstream changes
git fetch upstream-rw

# Rebase
git rebase upstream-rw/master
```

Remember that, if you do a rebase, and push that, you'll have to close any github pull requests manually, because github will not be able to detect the changes have already been merged.

A long series of commits

If there are a longer series of related commits, consider a merge instead:

```
git fetch upstream-rw
git merge --no-ff upstream-rw/master
```

The merge will be detected by github, and should close any related pull requests automatically.

Note the `--no-ff` above. This forces git to make a merge commit, rather than doing a fast-forward, so that these set of commits branch off trunk then rejoin the main history with a merge, rather than appearing to have been made directly on top of trunk.

Check the history

Now, in either case, you should check that the history is sensible and you have the right commits:

```
git log --oneline --graph
git log -p upstream-rw/master..
```

The first line above just shows the history in a compact way, with a text representation of the history graph. The second line shows the log of commits excluding those that can be reached from trunk (`upstream-rw/master`), and including those that can be reached from current HEAD (implied with the `..` at the end). So, it shows the commits unique to this branch compared to trunk. The `-p` option shows the diff for these commits in patch form.

Push to trunk

```
git push upstream-rw my-new-feature:master
```

This pushes the `my-new-feature` branch in this repository to the `master` branch in the `upstream-rw` repository.

Using Milestones and Labels

These guidelines are adapted from similar guidelines followed by IPython:

- 100% of confirmed issues and new features should have a milestone
- Only the following criteria should result in an issue being closed without a milestone:
  - Not actually an issue (user error, etc.)
  - Duplicate of an existing issue
  - A pull request superseded by a new pull request providing an alternate implementation
- Open issues should only lack a milestone if:
  - More clarification is required
  - Which milestone it belongs in requires some discussion
- Corollary: When an issue is closed without a milestone that means that the issue will not be fixed, or that it was not a real issue at all.
- In general there should be the following open milestones:

  - The next bug fix releases for any still-supported version lines; for example if 0.4 is in development and 0.2.x and 0.3.x are still supported there should be milestones for the next 0.2.x and 0.3.x releases.
  - The next X.Y release, i.e. the next minor release; this is generally the next release that all development in master is aimed toward.
  - The next X.Y release +1; for example if 0.3 is the next release, there should also be a milestone for 0.4 for issues that are important, but that we know won't be resolved in the next release.
  - Future–this is for all issues that require attention at some point but for which no immediate solution is in sight.
- Bug fix release milestones should only be used for deferring issues that won't be fixed in the next minor release, or for issues is previous releases that no longer apply to the mainline.
- When in doubt about which milestone to use for an issue, use the next minor release–it can always be moved once it's been more closely reviewed prior to release.
- Active milestones associated with a specific release (eg. v0.3.0) should contain at least one issue with the release label representing the actual task for releasing that version (this also works around the GitHub annoyance that milestones without any open issues are automatically closed).
- Issues that require fixing in the mainline, but that also are confirmed to apply to supported stable version lines should be marked with one or more `'backport-*'` labels for each v0.X.Y branch that has the issue.
  - In some cases it may require extra work beyond a simple merge to port bug fixes to older lines of development; if such additional work is required it is not a bad idea to open a "Backport #nnn to v0.X.Y" issue in the appropriate v0.X.Y milestone.

Updating and Maintaining the Changelog

The Astropy "changelog" is kept in the file `CHANGES.rst` at the root of the repository. As the filename extension suggests this is a reStructured Text file. The purpose of this file is to give a technical, but still user (and developer) oriented overview of what changes were made to Astropy between each public release. The idea is that it's a little more to the point and easier to follow than trying to read through full git log. It lists all new features added between versions, so that a user can easily find out from reading the changelog when a feature was added. Likewise it lists any features or APIs that were changed (and how they were changed) or removed. It also lists all bug fixes. Affiliated packages are encouraged to maintain a similar changelog.

Adding to the changelog

There are two approaches one may take to adding a new entry to the changelog, each with certain pros and cons. Before describing the two specific approaches it should be said that *all* additions to the changelog should be made first in the ‘master’ branch. This is because every release of Astropy includes a copy of the changelog, and it should list all the changes in every prior version of Astropy. For example, when Astropy v0.3.0 is released, in addition to the changes new to that version the changelog should have all the changes from every v0.2.x version (and earlier) released up to that point.

Two approaches for including a changelog entry for a new feature or bug fix are:

- Include the changelog update in the same pull request as the change. That is, assuming this change is being made in a pull request it can include an accurate changelog update along with it.

  Pro: An addition to the changelog is just like any other documentation update, and should be part of any atomic change to the software. It can be pulled into master along with the rest of the change.

  Con: If many pull requests also include changelog updates, they can quickly conflict with each other and require rebasing. This is not difficult to resolve if the only conflict is in the changelog, but it can still be trouble especially for new contributors.

- Add to the changelog after a change has been merged to master, whether by pull request or otherwise.

  Pro: Largely escapes the merge conflict issue.

  Cons: Isn’t included “atomically” in the merge commit, making it more difficult to keep track of for backporting. Requires new contributors to either make a second pull request or have a developer with push access to the main repository make the commit.

The first approach is probably preferable, especially for core contributors. But the latter approach is acceptable as well.

Changelog format

The exact formatting of the changelog content is a bit loose for now (though it might become stricter if we want to develop more tools around the changelog). The format can be mostly inferred by looking at previous versions. Each release gets its own heading (using the `-` heading marker) with the version and release date. Releases still under development have `(unreleased)` as there is no release date yet.

There are generally up to three subheadings (using the `^` marker): “New Features”, “API Changes”, “Bug Fixes”, and “Other Changes and Additions”. The latter is mostly a catch-all for miscellaneous changes, though there’s no reason not to make up additional sub-headings if it seems appropriate.

Under each sub-heading, changes are typically grouped according to which sub-package they pertain to. Changes that apply to more than one sub-package or that only apply to support modules like `logging` or `utils` may go under a “Misc” group.

The actual texts of the changelog entries are typically just one to three sentences–they should be easy to glance over. Most entries end with a reference to an issue/pull request number in square brackets.

A single changelog entry may also reference multiple small changes. For example:

```
- Minor documentation fixes and restructuring.
  [#935, #967, #978, #1004, #1028, #1047]
```

Beyond that, the best advice for updating the changelog is just to look at existing entries for previous releases and copy the format.

How to create and maintain a Python package using the Astropy template

If you run into any problems, don’t hesitate to ask for help on the astropy-dev mailing list!

The package-template repository provides a template for Python packages. This package design mirrors the layout of the main Astropy repository, as well as reusing much of the helper code used to organize Astropy. See the package template documentation for instructions on using the package template.

Releasing a Python package

You can release a package using the steps given below. In these instructions, we assume that the release is made from a fresh clone of the remote “main” repository and not from a forked copy. We also assume that the changelog file is named `CHANGES.rst`, like for the astropy core package. If instead you use Markdown, then you should replace `CHANGES.rst` by `CHANGES.md` in the instructions.

Note

The instructions below assume that you do not make use of bug fix branches in your workflow. If you do wish to create a bug fix branch, we recommend that you read over the more complete astropy Release Procedures and adapt them for your package.

1. Make sure that Travis and any other continuous integration is passing.
2. Update the `CHANGES.rst` file to make sure that all the changes are listed, and update the release date, which should currently be set to `unreleased`, to the current date in `yyyy-mm-dd` format.
3. Update the version number in `setup.cfg` to the version you're about to release, without the `.dev` suffix (e.g. `0.1`). If your package uses setuptools_scm to manage version numbers, you can skip this step.
4. Run `git clean -fxd` to remove any untracked files (WARNING: this will permanently remove any files that have not been previously committed, so make sure that you don't need to keep any of these files).
5. At this point, the command to run to build the tar file will depend on whether your package has a `pyproject.toml` file or not. If it does not, then:

   ```
   python setup.py build sdist --format=gztar
   ```

   If it does, then first make sure the build package is installed and up-to-date:

   ```
   pip install build --upgrade
   ```

   then create the source distribution with:

   ```
   python -m build --sdist .
   ```

   All following instructions will assume you have `pyproject.toml`. If you do not use `pyproject.toml` yet, please see https://docs.astropy.org/en/v4.0.x/development/astropy-package-template.html instead.

   In both cases, make sure that generated file is good to go by going inside `dist`, expanding the tar file, going inside the expanded directory, and running the tests with:

   ```
   pip install -e .[test]
   pytest
   ```

   You may need to add the `--remote-data` flag or any other flags that you normally add when fully testing your package.
6. Go back to the root of the directory and remove the generated files with:

   ```
   git clean -fxd
   ```

7. Add the changes to `CHANGES.rst` and `setup.cfg`:

```
git add CHANGES.rst setup.cfg
```

and commit with message:

```
git commit -m "Preparing release <version>"
```

8. Tag commit with `v<version>`, optionally signing with the `-s` option:

```
git tag v<version>
```

9. Change `VERSION` in `setup.cfg` to next version number, but with a `.dev` suffix at the end (e.g. `0.2.dev`). Add a new section to `CHANGES.rst` for next version, with a single entry `No changes yet`, e.g.:

```
0.2 (unreleased)
----------------

- No changes yet
```

10. Add the changes to `CHANGES.rst` and `setup.cfg`:

```
git add CHANGES.rst setup.cfg
```

and commit with message:

```
git commit -m "Back to development: <next_version>"
```

11. Check out the release commit with `git checkout v<version>`. Run `git clean -fxd` to remove any non-committed files.
12. (optional) Run the tests in an environment that mocks up a “typical user” scenario. This is not strictly necessary because you ran the tests above, but it can sometimes be useful to catch subtle bugs that might come from you using a customized developer environment. For more on setting up virtual environments, see Python virtual environments, but for the sake of example we will assume you’re using Anaconda. Do:

```
conda create -n myaffilpkg_rel_test astropy <any more dependencies
here>
source activate myaffilpkg_rel_test
python -m build --sdist .
cd dist
pip install myaffilpkg-version.tar.gz
```

```
python -c 'import myaffilpkg; myaffilpkg.test()'
source deactivate
cd <back to your source>
```

You may want to repeat this for other combinations of dependencies if you think your users might have other relevant packages installed. Assuming the tests all pass, you can proceed on.

13. If you did the previous step, do `git clean -fxd` again to remove anything you made there. Run `python -m build --sdist .` to create the files for upload. Then you can upload to PyPI via `twine`:

```
twine upload dist/*
```

as described in these instructions. Check that the entry on PyPI is correct, and that the tarfile is present.

14. Go back to the master branch and push your changes to github:

```
git checkout master
git push --tags origin master
```

Once you have done this, if you use Read the Docs, trigger a `latest` build then go to the project settings, and under **Versions** you should see the tag you just pushed. Select the tag to activate it, and save.

15. If your package is available in the `conda-forge` conda channel, you should also submit a pull request to update the version number in the feedstock of your package.

Modifications for a beta/release candidate release

Before a new release of your package, you may wish do a "pre-release" of the code, for example to allow collaborators to independently test the release. If the release you are performing is this kind of pre-release, some of the above steps need to be modified.

The primary modifications to the release procedure is:

- When entering the new version number, instead of just removing the `.dev`, enter "1.2b1" or "1.2rc1". It is critical that you follow this numbering scheme (`X.Yb#` or `X.Y.Zrc#`), as it will ensure the release is ordered "before" the main release by various automated tools, and also tells PyPI that this is a "pre-release".

Full Changelog

4.2 (2020-11-24)

New Features

astropy.convolution

- Methods `convolve` and `convolve_fft` both now return Quantity arrays if user input is given in one. [#10822]

astropy.coordinates

- Numpy functions that broadcast, change shape, or index (like `np.broadcast_to`, `np.rot90`, or `np.roll`) now work on coordinates, frames, and representations. [#10337]
- Add a new science state `astropy.coordinates.erfa_astrom.erfa_astrom` and two classes `ErfaAstrom`, `ErfaAstromInterpolator` as wrappers to the `pyerfa` astrometric functions used in the coordinate transforms. Using `ErfaAstromInterpolator`, which interpolates astrometric properties for `SkyCoord` instances with arrays of obstime, can dramatically speed up coordinate transformations while keeping microarcsecond resolution. Depending on needed precision and the obstime array in question, speed ups reach factors of 10x to >100x. [#10647]
- `galactocentric_frame_defaults` can now also be used as a registry, with user-defined parameter values and metadata. [#10624]
- Method `.realize_frame` from coordinate frames now accepts `**kwargs`, including `representation_type`. [#10727]
- Avoid an unnecessary call to `erfa.epv00` in transformations between `CIRS` and `ICRS`, improving performance by 50 %. [#10814]
- A new equatorial coordinate frame, with RA and Dec measured w.r.t to the True Equator and Equinox (TETE). This frame is commonly known as “apparent place” and is the correct frame for coordinates returned from JPL Horizons. [#10867]
- Added a context manager `impose_finite_difference_dt` to the `TransformGraph` class to override the finite-difference time step attribute (`finite_difference_dt`) for all transformations in the graph with that attribute. [#10341]
- Improve performance of `SpectralCoord` by refactoring internal implementation. [#10398]

astropy.cosmology

- The final version of the Planck 2018 cosmological parameters are included as the `Planck18` object, which is now the default cosmology. The parameters are identical to those of the `Planck18_arXiv_v2` object, which is now deprecated and will be removed in a future release. [#10915]

astropy.modeling

- Added NFW profile and tests to modeling package [#10505]
- Added missing logic for evaluate to compound models [#10002]
- Stop iteration in `FittingWithOutlierRemoval` before reaching `niter` if the masked points are no longer changing. [#10642]
- Keep a (shallow) copy of `fit_info` from the last iteration of the wrapped fitter in `FittingWithOutlierRemoval` and also record the actual number of iterations performed in it. [#10642]
- Added attributes for fitting uncertainties (covariance matrix, standard deviations) to models. Parameter covariance matrix can be accessed via `model.cov_matrix`, standard deviations by `model.stds` or individually for each parameter by `parameter.std`. Currently implemented for `LinearLSQFitter` and `LevMarLSQFitter`. [#10552]
- N-dimensional least-squares statistic and specific 1,2,3-D methods [#10670]

astropy.stats

- Added `circstd` function to obtain a circular standard deviation. [#10690]

astropy.table

- Allow initializing a `Table` using a list of `names` in conjunction with a `dtype` from a numpy structured array. The list of `names` overrides the names specified in the `dtype`. [#10419]

astropy.time

- Add new `isclose()` method to `Time` and `TimeDelta` classes to allow comparison of time objects to within a specified tolerance. [#10646]
- Improve initialization time by a factor of four when creating a scalar `Time` object in a format like `unix` or `cxcsec` (time delta from a reference epoch time). [#10406]
- Improve initialization time by a factor of ~25 or more for large arrays of string times in ISO, ISOT or year day-of-year formats. This is done with a new C-based time parser that can be adapted for other fixed-format custom time formats. [#10360]

- Numpy functions that broadcast, change shape, or index (like `np.broadcast_to`, `np.rot90`, or `np.roll`) now work on times. [#10337, #10502]

astropy.timeseries

- Improve memory and speed performance when iterating over the entire time column of a `TimeSeries` object. Previously this involved O(N^2) operations and memory. [#10889]

astropy.units

- `Quantity.to` has gained a `copy` option to allow copies to be avoided when the units do not change. [#10517]
- Added the `spat` unit of solid angle that represents the full sphere. [#10726]

astropy.utils

- `ShapedLikeNDArray` has gained the capability to use numpy functions that broadcast, change shape, or index. [#10337]
- `get_free_space_in_dir` now takes a new `unit` keyword and `check_free_space_in_dir` takes `size` defined as `Quantity`. [#10627]
- New `astropy.utils.data.conf.allow_internet` configuration item to control downloading data from the Internet. Setting `allow_internet=False` is the same as `remote_timeout=0`. Using `remote_timeout=0` to control internet access will stop working in a future release. [#10632]
- New `is_url` function so downstream packages do not have to secretly use the hidden `_is_url` anymore. [#10684]

astropy.visualization

- Added the `Quadrangle` patch for `WCSAxes` for a latitude-longitude quadrangle. Unlike `matplotlib.patches.Rectangle`, the edges of this patch will be rendered as curved lines if appropriate for the WCS transformation. [#10862]
- The position of tick labels are now only calculated when needed. If any text parameters are changed (color, font weight, size etc.) that don’t effect the tick label position, the positions are not recomputed, improving performance. [#10806]

astropy.wcs

- `WCS.to_header()` now appends comments to SIP coefficients. [#10480]

- A new property `dropped_world_dimensions` has been added to `SlicedLowLevelWCS` to record information about any world axes removed by slicing a WCS. [#10195]
- New `WCS.proj_plane_pixel_scales()` and `WCS.proj_plane_pixel_area()` methods to return pixel scales and area, respectively, as Quantity. [#10872]

API Changes

astropy.config

- `set_temp_config` now preserves the existing cache rather than deleting it and relying on reloading it from the previous config file. This ensures that any programmatically made changes are preserved as well. [#10474]
- Configuration path detection logic has changed: Now, it looks for `~` first before falling back to older logic. In addition, `HOMESHARE` is no longer used in Windows. [#10705]

astropy.coordinates

- The passing of frame classes (as opposed to frame instances) to the `transform_to()` methods of low-level coordinate-frame classes has been deprecated. Frame classes can still be passed to the `transform_to()` method of the high-level `SkyCoord` class, and using `SkyCoord` is recommended for all typical use cases of transforming coordinates. [#10475]

astropy.stats

- Added a `grow` parameter to `SigmaClip`, `sigma_clip` and `sigma_clipped_stats`, to allow expanding the masking of each deviant value to its neighbours within a specified radius. [#10613]
- Passing float `n` to `poisson_conf_interval` when using `interval='kraft-burrows-nousek'` now raises `TypeError` as its value must be an integer. [#10838]

astropy.table

- Change `Table.columns.keys()` and `Table.columns.values()` to both return generators instead of a list. This matches the behavior for Python `dict` objects. [#10543]
- Removed the `FastBST` and `FastRBT` indexing engines because they depend on the `bintrees` package, which is no longer maintained and is deprecated. Instead, use the `SCEngine` indexing engine, which is similar in

performance and relies on the `sortedcontainers` package. [#10622]

- When slicing a mixin column in a table that had indices, the indices are no longer copied since they generally are not useful, having the wrong shape. With this, the behaviour becomes the same as that for a regular `Column`. (Note that this does not affect slicing of a table; sliced columns in those will continue to carry a sliced version of any indices). [#10890]
- Change behavior so that when getting a single item out of a mixin column such as `Time`, `TimeDelta`, `SkyCoord` or `Quantity`, the `info` attribute is no longer copied. This improves performance, especially when the object is an indexed column in a `Table`. [#10889]
- Raise a TypeError when a scalar column is added to an unsized table. [#10476]
- The order of columns when creating a table from a `list` of `dict` may be changed. Previously, the order was alphabetical because the `dict` keys were assumed to be in random order. Since Python 3.7, the keys are always in order of insertion, so `Table` now uses the order of keys in the first row to set the column order. To alphabetize the columns to match the previous behavior, use `t = t[sorted(t.colnames)]`. [#10900]

astropy.time

- Refactor `Time` and `TimeDelta` classes to inherit from a common `TimeBase` class. The `TimeDelta` class no longer inherits from `Time`. A number of methods that only apply to `Time` (e.g. `light_travel_time`) are no longer available in the `TimeDelta` class. [#10656]

astropy.units

- The `bar` unit is no longer wrongly considered an SI unit, meaning that SI decompositions like `(u.kg*u.s**-2* u.sr**-1 * u.nm**-1).si` will no longer include it. [#10586]

astropy.utils

- Shape-related items from `astropy.utils.misc` – `ShapedLikeNDArray`, `check_broadcast`, `unbroadcast`, and `IncompatibleShapeError` – have been moved to their own module, `astropy.utils.shapes`. They remain importable from `astropy.utils`. [#10337]
- `check_hashes` keyword in `check_download_cache` is deprecated and will be removed in a future release. [#10628]
- `hexdigest` keyword in `import_file_to_cache` is deprecated and will be removed in a future release. [#10628]

Bug Fixes

astropy.config

- Fix a few issues with `generate_config` when used with other packages. [#10893]

astropy.coordinates

- Fixed a bug in the coordinate-frame attribute `CoordinateAttribute` where the internal transformation could behave differently depending on whether the input was a low-level coordinate frame or a high-level `SkyCoord`. `CoordinateAttribute` now always performs a `SkyCoord`-style internal transformation, including the by-default merging of frame attributes. [#10475]

astropy.modeling

- Fixed an issue of `Model.render` when the input `out` datatype is not float64. [#10542]

astropy.visualization

- Fix support for referencing WCSAxes coordinates by their world axes names. [#10484]

astropy.wcs

- Objective functions called by `astropy.wcs.fit_wcs_from_points` were treating longitude and latitude distances equally. Now longitude scaled properly. [#10759]

Other Changes and Additions

- Minimum version of required Python is now 3.7. [#10900]
- Minimum version of required Numpy is now 1.17. [#10664]
- Minimum version of required Scipy is now 1.1. [#10900]
- Minimum version of required PyYAML is now 3.13. [#10900]
- Minimum version of required Matplotlib is now 3.0. [#10900]
- The private `_erfa` module has been converted to its own package, `pyerfa`, which is a required dependency for astropy, and can be imported with `import erfa`. Importing `_erfa` from `astropy` will give a deprecation warning. [#10329]
- Added `optimize=True` flag to calls of `yacc.yacc` (as already done for `lex.lex`) to allow running in `python -OO` session without raising an

exception in `astropy.units.format`. [#10379]

- Shortened FITS comment strings for some D2IM and CPDIS FITS keywords to reduce the number of FITS `VerifyWarning` warnings when working with WCSes containing lookup table distortions. [#10513]
- When importing astropy without first building the extension modules first, raise an error directly instead of trying to auto-build. [#10883]

4.1 (2020-10-21)

New Features

astropy.config

- Add new function `generate_config` to generate the configuration file and include it in the documentation. [#10148]
- `ConfigNamespace.__iter__` and `ConfigNamespace.keys` now yield `ConfigItem` names defined within it. Similarly, `items` and `values` would yield like a Python dictionary would. [#10139]

astropy.coordinates

- Added a new `SpectralCoord` class that can be used to define spectral coordinates and transform them between different velocity frames. [#10185]
- Angle parsing now supports `cardinal direction` in the cases where angles are initialized as `string` instances. eg `"17°53'27"W"`.[#9859]
- Allow in-place modification of array-valued `Frame` and `SkyCoord` objects. This provides limited support for updating coordinate data values from another coordinate object of the same class and equivalent frame attributes. [#9857]
- Added a robust equality operator for comparing `SkyCoord`, frame, and representation objects. A comparison like `sc1 == sc2` will now return a boolean or boolean array where the objects are strictly equal in all relevant frame attributes and coordinate representation values. [#10154]
- Added the True Equator Mean Equinox (TEME) frame. [#10149]
- The `Galactocentric` frame will now use the “latest” parameter definitions by default. This currently corresponds to the values defined in v4.0, but will change with future releases. [#10238]
- The `SkyCoord.from_name()` and Sesame name resolving functionality now is able to cache results locally and will do so by default. [#9162]
- Allow in-place modification of array-valued `Representation` and `Differential` objects, including of representations with attached differentials. [#10210]

astropy.io.ascii

- Functional Units can now be processed in CDS-tables. [#9971]
- Allow reading in ASCII tables which have duplicate column names. [#9939]
- Fixed failure of ASCII `fast_reader` to handle `names`, `include_names`, `exclude_names` arguments for `RDB` formatted tables. Homogenised checks and exceptions for invalid `names` arguments. Improved performance when parsing “wide” tables with many columns. [#10306]
- Added type validation of key arguments in calls to `io.ascii.read()` and `io.ascii.write()` functions. [#10005]

astropy.io.misc

- Added serialization of parameter constraints fixed and bounds. [#10082]
- Added ‘functional_models.py’ and ‘physical_models.py’ to asdf/tags /transform, with to allow serialization of all functional and physical models. [#10028, #10293]
- Fix ASDF serialization of circular model inverses, and remove explicit calls to `asdf.yamlutil` functions that became unnecessary in asdf 2.6.0. [#10189, #10384]

astropy.io.fits

- Added support for writing Dask arrays to disk efficiently for `ImageHDU` and `PrimaryHDU`. [#9742]
- Add HDU name and ver to FITSDiff report where appropriate [#10197]

astropy.io.votable

- New `exceptions.conf.max_warnings` configuration item to control the number of times a type of warning appears before being suppressed. [#10152]
- No longer ignore attributes whose values were specified as empty strings. [#10583]

astropy.modeling

- Added Plummer1D model to `functional_models`. [#9896]
- Added `UnitsMapping` model and `Model.coerce_units` to support units on otherwise unitless models. [#9936]
- Added `domain` and `window` attributes to `repr` and `str`. Fixed bug with `_format_repr` in core.py. [#9941]
- Polynomial attributes `domain` and `window` are now tuples of size 2 and

are validated. `repr` and `print` show only their non-default values. [#10145]

- Added `replace_submodel()` method to `CompoundModel` to modify an existing instance. [#10176]
- Delay construction of `CompoundModel` inverse until property is accessed, to support ASDF deserialization of circular inverses in component models. [#10384]

astropy.nddata

- Added support in the `bitmask` module for using mnemonic bit flag names when specifying the bit flags to be used or ignored when converting a bit field to a boolean. [#10095, #10208]
- Added `reshape_as_blocks` function to reshape a data array into blocks, which is useful to efficiently apply functions on block subsets of the data instead of using loops. The reshaped array is a view of the input data array. [#10214]
- Added a `cache` keyword option to allow caching for `CCDData.read` if filename is a URL. [#10265]

astropy.table

- Added ability to specify a custom matching function for table joins. In particular this makes it possible to do cross-match table joins on `SkyCoord`, `Quantity`, or standard columns, where column entries within a specified distance are considered to be matched. [#10169]
- Added `units` and `descriptions` keyword arguments to the Table object initialization and `Table.read()` methods. This allows directly setting the `unit` and `description` for the table columns at the time of creating or reading the table. [#9671]
- Make table `Row` work as mappings, by adding `.keys()` and `.values()` methods. With this `**row` becomes possible, as does, more simply, turning a `Row` into a dictionary with `dict(row)`. [#9712]
- Added two new `Table` methods `.items()` and `.values()`, which return respectively `tbl.columns.items()` (iterator over name, column tuples) and `tbl.columns.values()` (list of columns) for a `Table` object `tbl`. [#9780]
- Added new `Table` method `.round()`, which rounds numeric columns to the specified number of decimals. [#9862]
- Updated `to_pandas()` and `from_pandas()` to use and support Pandas nullable integer data type for masked integer data. [#9541]
- The HDF5 writer, `write_table_hdf5()`, now allows passing through additional keyword arguments to the `h5py.Group.create_dataset()`.

[#9602]
- Added capability to add custom table attributes to a `Table` subclass. These attributes are persistent and can be set during table creation. [#10097]
- Added support for `SkyCoord` mixin columns in `dstack`, `vstack` and `insert_row` functions. [#9857]
- Added support for coordinate `Representation` and `Differential` mixin columns. [#10210]

astropy.time

- Added a new time format `unix_tai` which is essentially Unix time but with leap seconds included. More precisely, this is the number of seconds since `1970-01-01 00:00:08 TAI` and corresponds to the `CLOCK_TAI` clock available on some linux platforms. [#10081]

astropy.units

- Added `torr` pressure unit. [#9787]
- Added the `equal_nan` keyword argument to `isclose` and `allclose`, and updated the docstrings. [#9849]
- Added `Rankine` temperature unit. [#9916]
- Added integrated flux unit conversion to `spectral_density` equivalency. [#10015]
- Changed `pixel_scale` equivalency to allow scales defined in any unit. [#10123]
- The `quantity_input` decorator now optionally allows passing through numeric values or numpy arrays with numeric dtypes to arguments where `dimensionless_unscaled` is an allowed unit. [#10232]

astropy.utils

- Added a new `MetaAttribute` class to support easily adding custom attributes to a subclass of classes like `Table` or `NDData` that have a `meta` attribute. [#10097]

astropy.visualization

- Added `invalid` keyword to `SqrtStretch`, `LogStretch`, `PowerStretch`, and `ImageNormalize` classes and the `simple_norm` function. This keyword is used to replace generated NaN values. [#10182]
- Fixed an issue where ticks were sometimes not drawn at the edges of a spherical projection on a WCSAxes. [#10442]

astropy.wcs

- WCS objects with a spectral axis will now return `SpectralCoord` objects when calling `pixel_to_world` instead of `Quantity`, and can now take either `Quantity` or `SpectralCoord` as input to `pixel_to_world`. [#10185]
- Implemented support for the `-TAB` algorithm (WCS Paper III). [#9641]
- Added an `_as_mpl_axes` method to the `HightLevelWCSWrapper` class. [#10138]
- Add .upper() to ctype or ctype names to wcsapi/fitwcs.py to mitigate bugs from unintended lower/upper case issues [#10557]

API Changes

astropy.coordinates

- The equality operator for comparing `SkyCoord`, frame, and representation objects was changed. A comparison like `sc1 == sc2` was previously equivalent to `sc1 is sc2`. It will now return a boolean or boolean array where the objects are strictly equal in all relevant frame attributes and coordinate representation values. If the objects have different frame attributes or representation types then an exception will be raised. [#10154]
- ``` `SkyCoord.radial_velocity_correction` ``` now allows you to pass an ``` `obstime` ``` directly when the ``` `SkyCoord` ``` also has an ``` `obstime` ``` set. In this situation, the position of the ``` `SkyCoord` ``` has space motion applied to correct to the passed ``` `obstime` ```. This allows mm/s radial velocity precision for objects with large space motion. [#10094]
- For consistency with other astropy classes, coordinate `Representations` and `Differentials` can now be initialized with an instance of their own class if that instance is passed in as the first argument. [#10210]

astropy.io.ascii

- Changed the behavior when reading a table where both the `names` argument is provided (to specify the output column names) and the `converters` argument is provided (to specify column conversion functions). Previously the `converters` dict names referred to the *input* table column names, but now they refer to the *output* table column names. [#9739]

astropy.io.votable

- For FIELDs with datatype=”char”, store the values as strings instead of bytes. [#9505]

astropy.table

- `Table.from_pandas` now supports a `units` dictionary as argument to pass units for columns in the `DataFrame`. [#9472]

astropy.time

- Require that `in_subfmt` and `out_subfmt` properties of a `Time` object have allowed values at the time of being set, either when creating the object or when setting those properties on an existing `Time` instance. Previously the validation of those properties was not strictly enforced. [#9868]

astropy.utils

- Changed the exception raised by `get_readable_fileobj` on missing compression modules (for `bz2` or `lzma`/`xz` support) to `ModuleNotFoundError`, consistent with `io.fits` file handlers. [#9761]

astropy.visualization

- Deprecated the `imshow_only_kwargs` keyword in `imshow_norm`. [#9915]
- Non-finite input values are now automatically excluded in `HistEqStretch` and `InvertedHistEqStretch`. [#10177]
- The `PowerDistStretch` and `InvertedPowerDistStretch` `a` value is restricted to be `a >= 0` in addition to `a != 1`. [#10177]
- The `PowerStretch`, `LogStretch`, and `InvertedLogStretch` `a` value is restricted to be `a > 0`. [#10177]
- The `AsinhStretch` and `SinhStretch` `a` value is restricted to be `0 < a <= 1`. [#10177]

Bug Fixes

astropy.coordinates

- Fix a bug where for light deflection by the Sun it was always assumed that the source was at infinite distance, which in the (rare and) absolute worst-case scenario could lead to errors up to 3 arcsec. [#10666]

astropy.io.votable

- For FIELDs with datatype=”char”, store the values as strings instead of bytes. [#9505]

astropy.table

- Fix a bug that prevented `Time` columns from being used to sort a table. [#10824]

astropy.wcs

- WCS objects with a spectral axis will now return `SpectralCoord` objects when calling `pixel_to_world` instead of `Quantity` (note that `SpectralCoord` is a sub-class of `Quantity`). [#10185]
- Add .upper() to ctype or ctype names to wcsapi/fitwcs.py to mitigate bugs from unintended lower/upper case issues [#10557]
- Added bounds to `fit_wcs_from_points` to ensure CRPIX is on input image. [#10346]

Other Changes and Additions

- The way in which users can specify whether to build astropy against existing installations of C libraries rather than the bundled one has changed, and should now be done via environment variables rather than setup.py flags (e.g. –use-system-erfa). The available variables are `ASTROPY_USE_SYSTEM_CFITSIO`, `ASTROPY_USE_SYSTEM_ERFA`, `ASTROPY_USE_SYSTEM_EXPAT`, `ASTROPY_USE_SYSTEM_WCSLIB`, and `ASTROPY_USE_SYSTEM_ALL`. These should be set to `1` to build against the system libraries. [#9730]
- The infrastructure of the package has been updated in line with the APE 17 roadmap (https://github.com/astropy/astropy-APEs/blob/master/APE17.rst). The main changes are that the `python setup.py test` and `python setup.py build_docs` commands will no longer work. The easiest way to replicate these commands is to install the tox (https://tox.readthedocs.io) package and run `tox -e test` and `tox -e build_docs`. It is also possible to run pytest and sphinx directly. Other significant changes include switching to setuptools_scm to manage the version number, and adding a `pyproject.toml` to opt in to isolated builds as described in PEP 517/518. [#9726]
- Bundled `expat` is updated to version 2.2.9. [#10038]
- Increase minimum asdf version to 2.6.0. [#10189]
- The bundled version of PLY was updated to 3.11. [#10258]
- Removed dependency on scikit-image. [#10214]

4.0.4 (2020-11-24)

Bug Fixes

astropy.coordinates

- The `norm()` method for `RadialDifferential` no longer requires `base` to be specified. The `norm()` method for other non-Cartesian differential classes now gives a clearer error message if `base` is not specified. [#10969]
- The transformations between `ICRS` and any of the heliocentric ecliptic frames (`HeliocentricMeanEcliptic`, `HeliocentricTrueEcliptic`, and `HeliocentricEclipticIAU76`) now correctly account for the small motion of the Sun when transforming a coordinate with velocity information. [#10970]

astropy.io.ascii

- Partially fixed a performance issue when reading in parallel mode. Parallel reading currently has substantially worse performance than the default serial reading, so we now ignore the parallel option and fall back to serial reading. [#10880]
- Fixed a bug where “” (blank string) as input data for a boolean type column was causing an exception instead of indicating a masked value. As a consequence of the fix, the values “0” and “1” are now also allowed as valid inputs for boolean type columns. These new allowed values apply for both ECSV and for basic character-delimited data files (‘basic’ format with appropriate `converters` specified). [#10995]

astropy.modeling

- Fixed use of weights with `LinearLSQFitter`. [#10687]

astropy.stats

- Fixed an issue in biweight stats when MAD=0 to give the same output with and without an input `axis`. [#10912]

astropy.time

- Fix a problem with the `plot_date` format for matplotlib >= 3.3 caused by a change in the matplotlib plot date default reference epoch in that release. [#10876]
- Improve initialization time by a factor of four when creating a scalar `Time` object in a format like `unix` or `cxcsec` (time delta from a reference epoch time). [#10406]

astropy.visualization

- Fixed the caclulation of the tight bounding box of a `WCSAxes`. This should also significantly improve the application of `tight_layout()` to figures containing `WCSAxes`. [#10797]

4.0.3 (2020-10-14)

Bug Fixes

astropy.table

- Fixed a small bug where initializing an empty `Column` with a structured dtype with a length and a shape failed to give the requested dtype. [#10819]

Other Changes and Additions

- Fixed installation of the source distribution with pip<19. [#10837, #10852]

4.0.2 (2020-10-10)

New Features

astropy.utils

- `astropy.utils.data.download_file` now supports FTPS/FTP over TLS. [#9964]
- `astropy.utils.data` now uses a lock-free mechanism for caching. This new mechanism uses a new cache layout and so ignores caches created using earlier mechanisms (which were causing lockups on clusters). The two cache formats can coexist but do not share any files. [#10437, #10683]
- `astropy.utils.data` now ignores the config item `astropy.utils.data.conf.download_cache_lock_attempts` since no locking is done. [#10437, #10683]
- `astropy.utils.data.download_file` and related functions now interpret the parameter or config file setting `timeout=0` to mean they should make no attempt to download files. [#10437, #10683]
- `astropy.utils.import_file_to_cache` now accepts a keyword-only argument `replace`, defaulting to True, to determine whether it should replace existing files in the cache, in a way as close to atomic as possible. [#10437, #10683]
- `astropy.utils.data.download_file` and related functions now treat `http://example.com` and `http://example.com/` as equivalent. [#10631]

astropy.wcs

- The new auxiliary WCS parameters added in WCSLIB 7.1 are now exposed as the `aux` attribute of `Wcsprm`. [#10333]
- Updated bundled version of `WCSLIB` to v7.3. [#10433]

Bug fixes

astropy.config

- Added an extra fallback to `os.expanduser('~')` when trying to find the user home directory. [#10570]

astropy.constants

- Corrected definition of parsec to 648 000 / pi AU following IAU 2015 B2 [#10569]

astropy.convolution

- Fixed a bug where a float-typed integers in the argument `x_range` of `astropy.convolution.utils.discretize_oversample_1D` (and the 2D version as well) fails because it uses `numpy.linspace`, which requires an `int`. [#10696]

astropy.coordinates

- Ensure that for size-1 array `SkyCoord` and coordinate frames the attributes also properly become scalars when indexed with 0. [#10113]
- Fixed a bug where `SkyCoord.separation()` and `SkyCoord.separation_3d` were not accepting a frame object. [#10332]
- Ensure that the `lon` values in `SkyOffsetFrame` are wrapped correctly at 180 degree regardless of how the underlying data is represented. [#10163]
- Fixed an error in the obliquity of the ecliptic when transforming to/from the `*TrueEcliptic` coordinate frames. The error would primarily result in an inaccuracy in the ecliptic latitude on the order of arcseconds. [#10129]
- Fixed an error in the computation of the location of solar system bodies where the Earth location of the observer was ignored during the correction for light travel time. [#10292]
- Ensure that coordinates with proper motion that are transformed to other coordinate frames still can be represented properly. [#10276]
- Improve the error message given when trying to get a cartesian representation for coordinates that have both proper motion and radial velocity, but no distance. [#10276]

- Fixed an error where `SkyCoord.apply_space_motion` would return incorrect results when no distance is set and proper motion is high. [#10296]
- Make the parsing of angles thread-safe so that `Angle` can be used in Python multithreading. [#10556]
- Fixed reporting of `EarthLocation.info` which previously raised an exception. [#10592]

astropy.io.ascii

- Fixed a bug with the C `fast_reader` not correctly parsing newlines when `delimiter` was also set to `\n` or `\r`; ensured consistent handling of input strings without newline characters. [#9929]

astropy.io.fits

- Fix integer formats of `TFORMn=Iw` columns in ASCII tables to correctly read values exceeding int32 - setting int16, int32 or int64 according to `w`. [#9901]
- Fix unclosed memory-mapped FITS files in `FITSDiff` when difference found. [#10159]
- Fix crash when reading an invalid table file. [#10171]
- Fix duplication issue when setting a keyword ending with space. [#10482]
- Fix ResourceWarning with `fits.writeto` and `pathlib.Path` object. [#10599]
- Fix repr for commentary cards and strip spaces for commentary keywords. [#10640]
- Fix compilation of cfitsio with Xcode 12. [#10772]
- Fix handling of 1-dimensional arrays with a single element in `BinTableHDU` [#10768]

astropy.io.misc

- Fix id URL in `baseframe-1.0.0` ASDF schema. [#10223]
- Write keys to ASDF only if the value is present, to account for a change in behavior in asdf 2.8. [#10674]

astropy.io.registry

- Fix `Table.(read|write).help` when reader or writer has no docstring. [#10460]

astropy.io.votable

- Fixed parsing failure of VOTable with no fields. When detecting a non-empty table with no fields, the following warning/exception is issued: E25 “No FIELDs are defined; DATA section will be ignored.” [#10192]

astropy.modeling

- Fixed a problem with mapping `input_units` and `return_units` of a `CompoundModel` to the units of the constituent models. [#10158]
- Removed hard-coded names of inputs and outputs. [#10174]
- Fixed a problem where slicing a `CompoundModel` by name will crash if there `fix_inputs` operators are present. [#10224]
- Removed a limitation of fitting of data with units with compound models without units when the expression involves operators other than addition and subtraction. [#10415]
- Fixed a problem with fitting `Linear1D` and `Planar2D` in model sets. [#10623]
- Fixed reported module name of `math_functions` model classes. [#10694]
- Fixed reported module name of `tabular` model classes. [#10709]
- Do not create new `math_functions` models for ufuncs that are only aliases (divide and mod). [#10697]
- Fix calculation of the `Moffat2D` derivative with respect to gamma. [#10784]

astropy.stats

- Fixed an API regression where `SigmaClip.__call__` would convert masked elements to `nan` and upcast the dtype to `float64` in its output `MaskedArray` when using the `axis` parameter along with the defaults `masked=True` and `copy=True`. [#10610]
- Fixed an issue where fully masked `MaskedArray` input to `sigma_clipped_stats` gave incorrect results. [#10099]
- Fixed an issue where `sigma_clip` and `SigmaClip.__call__` would return a masked array instead of a `ndarray` when `masked=False` and the input was a full-masked `MaskedArray`. [#10099]
- Fixed bug with `funcs.poisson_conf_interval` where an integer for N with `interval='kraft-burrows-nousek'` would throw an error with mpmath backend. [#10427]
- Fixed bug in `funcs.poisson_conf_interval` with `interval='kraft-burrows-nousek'` where certain combinations of source and background count numbers led to `ValueError` due to the choice of starting value for numerical optimization. [#10618]

astropy.table

- Fixed a bug when writing a table with mixin columns to FITS, ECSV or

HDF5. If one of the data attributes of the mixin (e.g. `skycoord.ra`) had the same name as one of the table column names (`ra`), the column (`ra`) would be dropped when reading the table back. [#10222]
- Fixed a bug when sorting an indexed table on the indexed column after first sorting on another column. [#10103]
- Fixed a bug in table argsort when called with `reverse=True` for an indexed table. [#10103]
- Fixed a performance regression introduced in #9048 when initializing a table from Python lists. Also fixed incorrect behavior (for data types other than float) when those lists contain `np.ma.masked` elements to indicate masked data. [#10636]
- Avoid modifying `.meta` when serializing columns to FITS. [#10485]
- Avoid crash when reading a FITS table that contains mixin info and PyYAML is missing. [#10485]

astropy.time

- Ensure that for size-1 array `Time`, the location also properly becomes a scalar when indexed with 0. [#10113]

astropy.units

- Refined test_parallax to resolve difference between 2012 and 2015 definitions. [#10569]

astropy.utils

- The default IERS server has been updated to use the FTPS server hosted by CDDIS. [#9964]
- Fixed memory allocation on 64-bit systems within `xml.iterparse` [#10076]
- Fix case where `None` could be used in a numerical computation. [#10126]

astropy.visualization

- Fixed a bug where the `ImageNormalize` `clip` keyword was ignored when used with calling the object on data. [#10098]
- Fixed a bug where `axes.xlabel`/`axes.ylabel` where not correctly set nor returned on an `EllipticalFrame` class `WCSAxes` plot. [#10446]

astropy.wcs

- Handled WCS 360 -> 0 deg crossover in `fit_wcs_from_points` [#10155]
- Do not issue `DATREF` warning when `MJDREF` has default value. [#10440]

- Fixed a bug due to which `naxis` argument was ignored if `header` was supplied during the initialization of a WCS object. [#10532]

Other Changes and Additions

- Improved the speed of sorting a large `Table` on a single column by a factor of around 5. [#10103]
- Ensure that astropy can be used inside Application bundles built with pyinstaller. [#8795]
- Updated the bundled CFITSIO library to 3.49. See `cextern/cfitsio/docs/changes.txt` for additional information. [#10256, #10665]
- `extract_array` raises a `ValueError` if the data type of the input array is inconsistent with the `fill_value`. [#10602]

4.0.1 (2020-03-27)

Bug fixes

astropy.config

- Fixed a bug where importing a development version of a package that uses `astropy` configuration system can result in a `~/.astropy/config/package..cfg` file. [#9975]

astropy.coordinates

- Fixed a bug where a vestigal trace of a frame class could persist in the transformation graph even after the removal of all transformations involving that frame class. [#9815]
- Fixed a bug with `TransformGraph.remove_transform()` when the “from” and “to” frame classes are not explicitly specified. [#9815]
- Read-only longitudes can now be passed in to `EarthLocation` even if they include angles outside of the range of -180 to 180 degrees. [#9900]
- `` `SkyCoord.radial_velocity_correction` `` no longer raises an Exception when space motion information is present on the SkyCoord. [#9980]

astropy.io

- Fixed a bug that prevented the unified I/O infrastructure from working with datasets that are represented by directories rather than files. [#9866]

astropy.io.ascii

- Fixed a bug in the `fast_reader` C parsers incorrectly returning entries of isolated positive/negative signs as `float` instead of `str`. [#9918]
- Fixed a segmentation fault in the `fast_reader` C parsers when parsing an invalid file with `guess=True` and the file contains inconsistent column numbers in combination with a quoted field; e.g., `"1  2\n 3  4 '5'"`. [#9923]
- Magnitude, decibel, and dex can now be stored in `ecsv` files. [#9933]

astropy.io.misc

- Magnitude, decibel, and dex can now be stored in `hdf5` files. [#9933]
- Fixed serialization of polynomial models to include non default values of domain and window values. [#9956, #9961]
- Fixed a bug which affected overwriting tables within `hdf5` files. Overwriting an existing path with associated column meta data now also overwrites the meta data associated with the table. [#9950]
- Fixed serialization of Time objects with location under time-1.0.0 ASDF schema. [#9983]

astropy.io.fits

- Fix regression with `GroupsHDU` which needs to modify the header to handle invalid headers, and fix accesing `.data` for empty HDU. [#9711, #9934]
- Fix `fitsdiff` when its arguments are directories that contain other directories. [#9711]
- Fix writing noncontiguous data to a compressed HDU. [#9958]
- Added verification of `disp` (`TDISP`) keyword to `fits.Column` and extended tests for `TFORM` and `TDISP` validation. [#9978]
- Fix checksum verification to process all HDUs instead of only the first one because of the lazy loading feature. [#10012]
- Allow passing `output_verify` to `.close` when using the context manager. [#10030]
- Prevent instantiation of `PrimaryHDU` and `ImageHDU` with a scalar. [#10041]
- Fix column access by attribute with FITS_rec: columns with scaling or columns from ASCII tables where not properly converted when accessed by attribute name. [#10069]

astropy.io.misc

- Magnitude, decibel, and dex can now be stored in `hdf5` files. [#9933]
- Fixed serialization of polynomial models to include non default values of

domain and window values. [#9956, #9961]
- Fixed a bug which affected overwriting tables within `hdf5` files. Overwriting an existing path with associated column meta data now also overwrites the meta data associated with the table. [#9950]
- Fixed serialization of Time objects with location under time-1.0.0 ASDF schema. [#9983]

astropy.modeling

- Fixed a bug in setting default values of parameters of orthonormal polynomials when constructing a model set. [#9987]

astropy.table

- Fixed bug in `Table.reverse` for tables that contain non-mutable mixin columns (like `SkyCoord`) for which in-place item update is not allowed. [#9839]
- Tables containing Magnitude, decibel, and dex columns can now be saved to `ecsv` files. [#9933]
- Fixed bug where adding or inserting a row fails on a table with an index defined on a column that is not the first one. [#10027]
- Ensured that `table.show_in_browser` also worked for mixin columns like `Time` and `SkyCoord`. [#10068]

astropy.time

- Fix inaccuracy when converting between TimeDelta and datetime.timedelta. [#9679]
- Fixed exception when changing `format` in the case when `out_subfmt` is defined and is incompatible with the new format. [#9812]
- Fixed exceptions in `Time.to_value()`: when supplying any `subfmt` argument for string-based formats like ‘iso’, and for `subfmt='long'` for the formats ‘byear’, ‘jyear’, and ‘decimalyear’. [#9812]
- Fixed bug where the location attribute was lost when creating a new `Time` object from an existing `Time` or list of `Time` objects. [#9969]
- Fixed a bug where an exception occurred when creating a `Time` object if the `val1` argument was a regular double and the `val2` argument was a `longdouble`. [#10034]

astropy.timeseries

- Fixed issue with reference time for the `transit_time` parameter returned by the `BoxLeastSquares` periodogram. Now, the `transit_time` will be within the range of the input data and arbitrary time offsets/zero points no longer affect results. [#10013]

astropy.units

- Fix for `quantity_input` annotation raising an exception on iterable types that don't define a general `__contains__` for checking if `None` is contained (e.g. Enum as of python3.8), by instead checking for instance of Sequence. [#9948]
- Fix for `u.Quantity` not taking into account `ndmin` if constructed from another `u.Quantity` instance with different but convertible unit [#10066]

astropy.utils

- Fixed `deprecated_renamed_argument` not passing in user value to deprecated keyword when the keyword has no new name. [#9981]
- Fixed `deprecated_renamed_argument` not issuing a deprecation warning when deprecated keyword without new name is passed in as positional argument. [#9985]
- Fixed detection of read-only filesystems in the caching code. [#10007]

astropy.visualization

- Fixed bug from matplotlib >=3.1 where an empty Quantity array is sent for unit conversion as an empty list. [#9848]
- Fix bug in `ZScaleInterval` to return the array minimum and maximum when there are less then `min_npixels` in the input array. [#9913]
- Fix a bug in simplifying axis labels that affected non-rectangular frames. [#8004, #9991]

Other Changes and Additions

- Increase minimum asdf version to 2.5.2. [#9996, #9819]
- Updated bundled version of `WCSLIB` to v7.2. [#10021]

4.0 (2019-12-16)

New Features

astropy.config

- The config and cache directories and the name of the config file are now customizable. This allows affiliated packages to put their configuration files in locations other than `CONFIG_DIR/.astropy/`. [#8237]

astropy.constants

- The version of constants can be specified via ScienceState in a way that `constants` and `units` will be consistent. [#8517]
- Default constants now use CODATA 2018 and IAU 2015 definitions. [#8761]
- Constants can be pickled and unpickled. [#9377]

astropy.convolution

- Fixed a bug [#9168] where having a kernel defined using unitless astropy quantity objects would result in a crash [#9300]

astropy.coordinates

- Changed `coordinates.solar_system_ephemeris` to also accept local files as input. The ephemeris can now be selected by either keyword (e.g. 'jpl', 'de430'), URL or file path. [#8767]
- Added a `cylindrical` property to `SkyCoord` for shorthand access to a `CylindricalRepresentation` of the coordinate, as is already available for other common representations. [#8857]
- The default parameters for the `Galactocentric` frame are now controlled by a `ScienceState` subclass, `galactocentric_frame_defaults`. New parameter sets will be added to this object periodically to keep up with ever-improved measurements of the solar position and motion. [#9346]
- Coordinate frame classes can now have multiple aliases by assigning a list of aliases to the class variable `name`. Any of the aliases can be used for attribute-style access or as the target of `tranform_to()` calls. [#8834]
- Passing a NaN to `Distance` no longer raises a warning. [#9598]

astropy.cosmology

- The pre-publication Planck 2018 cosmological parameters are included as the `Planck2018_arXiv_v2` object. Please note that the values are preliminary, and when the paper is accepted a final version will be included as `Planck18`. [#8111]

astropy.io.ascii

- Removed incorrect warnings on `Overflow` when reading in `FloatType` 0.0 with `use_fast_converter`; synchronised `IntType` `Overflow` warning messages. [#9082]

astropy.io.misc

- Eliminate deprecated compatibility mode when writing `Table` metadata to

HDF5 format. [#8899]
- Add support for orthogonal polynomial models to ASDF. [#9107]

astropy.io.fits

- Changed the `fitscheck` and `fitsdiff` script to use the `argparse` module instead of `optparse`. [#9148]
- Allow writing of `Table` objects with `Time` columns that are also table indices to FITS files. [#8077]

astropy.io.votable

- Support VOTable version 1.4. The main addition is the new element, TIMESYS, which allows defining of metadata for temporal coordinates much like COOSYS defines metadata for celestial coordinates. [#9475]

astropy.logger

- Added a configuration option to specify the text encoding of the log file, with the default behavior being the platform-preferred encoding. [#9203]

astropy.modeling

- Major rework of modeling internals. See modeling documentation for details. . [#8769]
- Add `Tabular1D.inverse`. [#9083]
- `Model.rename` was changed to add the ability to rename `Model.inputs` and `Model.outputs`. [#9220]
- New function `fix_inputs` to generate new models from others by fixing specific inputs variable values to constants. [#9135]
- `inputs` and `outputs` are now model instance attributes, and `n_inputs` and `n_outputs` are class attributes. Backwards compatible default values of `inputs` and `outputs` are generated. `Model.inputs` and `Model.outputs` are now settable which allows renaming them on per user case. [#9298]
- Add a new model representing a sequence of rotations in 3D around an arbitrary number of axes. [#9369]
- Add many of the numpy ufunc functions as models. [#9401]
- Add `BlackBody` model. [#9282]
- Add `Drude1D` model. [#9452]
- Added analytical King model (KingProjectedAnalytic1D). [#9084]
- Added Exponential1D and Logarithmic1D models. [#9351]

astropy.nddata

- Add a way for technically invalid but unambiguous units in a fits header to be parsed by `CCDData`. [#9397]
- `NDData` now only accepts WCS objects which implement either the high, or low level APE 14 WCS API. All WCS objects are converted to a high level WCS object, so `NDData.wcs` now always returns a high level APE 14 object. Not all array slices are valid for wcs objects, so some slicing operations which used to work may now fail. [#9067]

astropy.stats

- The `biweight_location`, `biweight_scale`, and `biweight_midvariance` functions now allow for the `axis` keyword to be a tuple of integers. [#9309]
- Added an `ignore_nan` option to the `biweight_location`, `biweight_scale`, and `biweight_midvariance` functions. [#9457]
- A numpy `MaskedArray` can now be input to the `biweight_location`, `biweight_scale`, and `biweight_midvariance` functions. [#9466]
- Removed the warning related to p0 in the Bayesian blocks algorithm. The caveat related to p0 is described in the docstring for `Events`. [#9567]

astropy.table

- Improved the implementation of `Table.replace_column()` to provide a speed-up of 5 to 10 times for wide tables. The method can now accept any input which convertible to a column of the correct length, not just `Column` subclasses. [#8902]
- Improved the implementation of `Table.add_column()` to provide a speed-up of 2 to 10 (or more) when adding a column to tables, with increasing benefit as the number of columns increases. The method can now accept any input which is convertible to a column of the correct length, not just `Column` subclasses. [#8933]
- Changed the implementation of `Table.add_columns()` to use the new `Table.add_column()` method. In most cases the performance is similar or slightly faster to the previous implemenation. [#8933]
- `MaskedColumn.data` will now return a plain `MaskedArray` rather than the previous (unintended) `masked_BaseColumn`. [#8855]
- Added depth-wise stacking `dstack()` in higher level table operation. It help will in stacking table column depth-wise. [#8939]
- Added a new table equality method `values_equal()` which allows comparison table values to another table, list, or value, and returns an element-by-element equality table. [#9068]
- Added new `join_type='cartesian'` option to the `join` operation.

[#9288]
- Allow adding a table column as a list of mixin-type objects, for instance `t['q'] = [1 * u.m, 2 * u.m]`. [#9165]
- Allow table `join()` using any sortable key column (e.g. Time), not just ndarray subclasses. A column is considered sortable if there is a `<column>.info.get_sortable_arrays()` method that is implemented. [#9340]
- Added `Table.iterrows()` for making row-wise iteration faster. [#8969]
- Allow table to be initialized with a list of dict where the dict keys are not the same in every row. The table column names are the set of all keys found in the input data, and any missing key/value pairs are turned into missing data in the table. [#9425]
- Prevent unnecessary ERFA warnings when indexing by `Time` columns. [#9545]
- Added support for sorting tables which contain non-mutable mixin columns (like `SkyCoord`) for which in-place item update is not allowed. [#9549]
- Ensured that inserting `np.ma.masked` (or any other value with a mask) into a `MaskedColumn` causes a masked entry to be inserted. [#9623]
- Fixed a bug that caused an exception when initializing a `MaskedColumn` from another `MaskedColumn` that has a structured dtype. [#9651]

astropy.tests

- The plugin that handles the custom header in the test output has been moved to the `pytest-astropy-header plugin` package. See the README at for information about using this new plugin. [#9214]

astropy.time

- Added a new time format `ymdhms` for representing times via year, month, day, hour, minute, and second attributes. [#7644]
- `TimeDelta` gained a `to_value` method, so that it becomes easier to use it wherever a `Quantity` with units of time could be used. [#8762]
- Made scalar `Time` and `TimeDelta` objects hashable based on JD, time scale, and location attributes. [#8912]
- Improved error message when bad input is used to initialize a `Time` or `TimeDelta` object and the format is specified. [#9296]
- Allow numeric time formats to be initialized with numpy `longdouble`, `Decimal` instances, and strings. One can select just one of these using `in_subfmt`. The output can be similarly set using `out_subfmt`. [#9361]
- Introduce a new `.to_value()` method for `Time` (and adjusted the existing method for `TimeDelta`) so that one can get values in a given

`format` and possible `subfmt` (e.g., `to_value('mjd', 'str')`. [#9361]
- Prevent unecessary ERFA warnings when sorting `Time` objects. [#9545]

astropy.timeseries

- Addig `epoch_phase`, `wrap_phase` and `normalize_phase` keywords to `TimeSeries.fold()` to control the phase of the epoch and to return normalized phase rather than time for the folded TimeSeries. [#9455]

astropy.uncertainty

- `Distribution` was rewritten such that it deals better with subclasses. As a result, Quantity distributions now behave correctly with `to` methods yielding new distributions of the kind expected for the starting distribution, and `to_value` yielding `NdarrayDistribution` instances. [#9429, #9442]
- The `pdf_*` properties that were used to calculate statistical properties of `Distrubution` instances were changed into methods. This allows one to pass parameters such as `ddof` to `pdf_std` and `pdf_var` (which generally should equal 1 instead of the default 0), and reflects that these are fairly involved calcuations, not just “properties”. [#9613]

astropy.units

- Support for unicode parsing. Currently supported are superscripts, Ohm, Ångström, and the micro-sign. [#9348]
- Accept non-unit type annotations in @quantity_input. [#8984]
- For numpy 1.17 and later, the new `__array_function__` protocol is used to ensure that all top-level numpy functions interact properly with `Quantity`, preserving units also in operations like `np.concatenate`. [#8808]
- Add equivalencies for surface brightness units to spectral_density. [#9282]

astropy.utils

- `astropy.utils.data.download_file` and `astropy.utils.data.get_readable_fileobj` now provides an `http_headers` keyword to pass in specific request headers for the download. It also now defaults to providing `User-Agent: Astropy` and `Accept: */*` headers. The default `User-Agent` value can be set with a new `astropy.data.conf.default_http_user_agent` configuration item. [#9508, #9564]
- Added a new `astropy.utils.misc.unbroadcast` function which can

be used to return the smallest array that can be broadcasted back to the initial array. [#9209]
- The specific IERS Earth rotation parameter table used for time and coordinate transformations can now be set, either in a context or per session, using `astropy.utils.iers.earth_rotation_table`. [#9244]
- Added `export_cache` and `import_cache` to permit transporting downloaded data to machines with no Internet connection. Several new functions are available to investigate the cache contents; e.g., `check_download_cache` can be used to confirm that the persistent cache has not become damaged. [#9182]
- A new `astropy.utils.iers.LeapSeconds` class has been added to track leap seconds. [#9365]

astropy.visualization

- Added a new `time_support` context manager/function for making it easy to plot and format `Time` objects in Matplotlib. [#8782]
- Added support for plotting any WCS compliant with the generalized (APE 14) WCS API with WCSAxes. [#8885, #9098]
- Improved display of information when inspecting `WCSAxes.coords`. [#9098]
- Improved error checking for the `slices=` argument to `WCSAxes`. [#9098]
- Added support for more solar frames in WCSAxes. [#9275]
- Add support for one dimensional plots to `WCSAxes`. [#9266]
- Add a `get_format_unit` to `wcsaxes.CoordinateHelper`. [#9392]
- `WCSAxes` now, by default, sets a default label for plot axes which is the WCS physical type (and unit) for that axis. This can be disabled using the `coords[i].set_auto_axislabel(False)` or by explicitly setting an axis label. [#9392]
- Fixed the display of tick labels when plotting all sky images that have a coord_wrap less than 360. [#9542]

astropy.wcs

- Added a `astropy.wcs.wcsapi.pixel_to_pixel` function that can be used to transform pixel coordinates in one dataset with a WCS to pixel coordinates in another dataset with a different WCS. This function is designed to be efficient when the input arrays are broadcasted views of smaller arrays. [#9209]
- Added a `local_partial_pixel_derivatives` function that can be used to determine a matrix of partial derivatives of each world coordinate with respect to each pixel coordinate. [#9392]

- Updated wcslib to v6.4. [#9125]
- Improved the `SlicedLowLevelWCS` class in `astropy.wcs.wcsapi` to avoid storing chains of nested `SlicedLowLevelWCS` objects when applying multiple slicing operations in turn. [#9210]
- Added a `wcs_info_str` function to `astropy.wcs.wcsapi` to show a summary of an APE-14-compliant WCS as a string. [#8546, #9207]
- Added two new optional attributes to the APE 14 low-level WCS: `pixel_axis_names` and `world_axis_names`. [#9156]
- Updated the WCS class to now correctly take and return `Time` objects in the high-level APE 14 API (e.g. `pixel_to_world`. [#9376]
- `SlicedLowLevelWCS` now raises `IndexError` rather than `ValueError` on an invalid slice. [#9067]
- Added `fit_wcs_from_points` function to `astropy.wcs.utils`. Fits a WCS object to set of matched detector/sky coordinates. [#9469]
- Fix various bugs in `SlicedLowLevelWCS` when the WCS being sliced was one dimensional. [#9693]

API Changes

astropy.constants

- Deprecated `set_enabled_constants` context manager. Use `astropy.physical_constants` and `astropy.astronomical_constants`. [#9025]

astropy.convolution

- Removed the deprecated keyword argument `interpolate_nan` from `convolve_fft`. [#9356]
- Removed the deprecated keyword argument `stddev` from `Gaussian2DKernel`. [#9356]
- Deprecated and renamed `MexicanHat1DKernel` and `MexicanHat2DKernel` to `RickerWavelet1DKernel` and `RickerWavelet2DKernel`. [#9445]

astropy.coordinates

- Removed the `recommended_units` attribute from Representations; it was deprecated since 3.0. [#8892]
- Removed the deprecated frame attribute classes, `FrameAttribute`, `TimeFrameAttribute`, `QuantityFrameAttribute`, `CartesianRepresentationFrameAttribute`; deprecated since 3.0. [#9326]

- Removed `longitude` and `latitude` attributes from `EarthLocation`; deprecated since 2.0. [#9326]
- The `DifferentialAttribute` for frame classes now passes through any input to the `allowed_classes` if only one allowed class is specified, i.e. this now allows passing a quantity in for frame attributes that use `DifferentialAttribute`. [#9325]
- Removed the deprecated `galcen_ra` and `galcen_dec` attributes from the `Galactocentric` frame. [#9346]

astropy.extern

- Remove the bundled `six` module. [#8315]

astropy.io.ascii

- Masked column handling has changed, see `astropy.table` entry below. [#8789]

astropy.io.misc

- Masked column handling has changed, see `astropy.table` entry below. [#8789]
- Removed deprecated `usecPickle` kwarg from `fnunpickle` and `fnpickle`. [#8890]

astropy.io.fits

- Masked column handling has changed, see `astropy.table` entry below. [#8789]
- `io.fits.Header` has been made safe for subclasses for copying and slicing. As a result of this change, the private subclass `CompImageHeader` now always should be passed an explicit `image_header`. [#9229]
- Removed the deprecated `tolerance` option in `fitsdiff` and `io.fits.diff` classes. [#9520]
- Removed deprecated keyword arguments for `CompImageHDU`: `compressionType`, `tileSize`, `hcompScale`, `hcompSmooth`, `quantizeLevel`. [#9520]

astropy.io.votable

- Changed `pedantic` argument to `verify` and change it to have three string-based options (`ignore`, `warn`, and `exception`) instead of just being a boolean. In addition, changed default to `ignore`, which means that warnings will not be shown by default when loading VO tables. [#8715]

astropy.modeling

- Eliminates support for compound classes (but not compound instances!) [#8769]
- Slicing compound models more restrictive. [#8769]
- Shape of parameters now includes n_models as dimension. [#8769]
- Parameter instances now hold values instead of models. [#8769]
- Compound model parameters now share instance and value with constituent models. [#8769]
- No longer possible to assign slices of parameter values to model parameters attribute (it is possible to replace it with a complete array). [#8769]
- Many private attributes and methods have changed (see documentation). [#8769]
- Deprecated `BlackBody1D` model and `blackbody_nu` and `blackbody_lambda` functions. [#9282]
- The deprecated `rotations.rotation_matrix_from_angle` was removed. [#9363]
- Deprecated and renamed `MexicanHat1D` and `MexicanHat2D` to `RickerWavelet1D` and `RickerWavelet2D`. [#9445]
- Deprecated `modeling.utils.ExpressionTree`. [#9576]

astropy.stats

- Removed the `iters` keyword from sigma clipping stats functions. [#8948]
- Renamed the `a` parameter to `data` in biweight stat functions. [#8948]
- Renamed the `a` parameter to `data` in `median_absolute_deviation`. [#9011]
- Renamed the `conflevel` keyword to `confidence_level` in `poisson_conf_interval`. Usage of `conflevel` now issues `AstropyDeprecationWarning`. [#9408]
- Renamed the `conf` keyword to `confidence_level` in `binom_conf_interval` and `binned_binom_proportion`. Usage of `conf` now issues `AstropyDeprecationWarning`. [#9408]
- Renamed the `conf_lvl` keyword to `confidence_level` in `jackknife_stats`. Usage of `conf_lvl` now issues `AstropyDeprecationWarning`. [#9408]

astropy.table

- The handling of masked columns in the `Table` class has changed in a way that may impact program behavior. Now a `Table` with `masked=False` may contain both `Column` and `MaskedColumn` objects, and adding a masked column or row to a table no longer “upgrades” the table and

columns to masked. This means that tables with masked data which are read via `Table.read()` will now always have `masked=False`, though specific columns will be masked as needed. Two new table properties `has_masked_columns` and `has_masked_values` were added. See the Masking change in astropy 4.0 section within for details. [#8789]
- Table operation functions such as `join`, `vstack`, `hstack`, etc now always return a table with `masked=False`, though the individual columns may be masked as necessary. [#8957]
- Changed implementation of `Table.add_column()` and `Table.add_columns()` methods. Now it is possible add any object(s) which can be converted or broadcasted to a valid column for the table. `Table.__setitem__` now just calls `add_column`. [#8933]
- Changed default table configuration setting `replace_warnings` from `['slice']` to `[]`. This removes the default warning when replacing a table column that is a slice of another column. [#9144]
- Removed the non-public method `astropy.table.np_utils.recarray_fromrecords`. [#9165]

astropy.tests

- In addition to `DeprecationWarning`, now `FutureWarning` and `ImportWarning` would also be turned into exceptions. [#8506]
- `warnings_to_ignore_by_pyver` option in `enable_deprecations_as_exceptions()` has changed. Please refer to API documentation. [#8506]
- Default settings for `warnings_to_ignore_by_pyver` are updated to remove very old warnings that are no longer relevant and to add a new warning caused by `pytest-doctestplus`. [#8506]

astropy.time

- `Time.get_ut1_utc` now uses the auto-updated `IERS_Auto` by default, instead of the bundled `IERS_B` file. [#9226]
- Time formats that do not use `val2` now raise ValueError instead of silently ignoring a provided value. [#9373]
- Custom time formats can now accept floating-point types with extended precision. Existing time formats raise exceptions rather than discarding extended precision through conversion to ordinary floating-point. [#9368]
- Time formats (implemented in subclasses of `TimeFormat`) now have their input and output routines more thoroughly validated, making it more difficult to create damaged `Time` objects. [#9375]
- The `TimeDelta.to_value()` method now can also take the `format`

name as its argument, in which case the value will be calculated using the `TimeFormat` machinery. For this case, one can also pass a `subfmt` argument to retrieve the value in another form than `float`. [#9361]

astropy.timeseries

- Keyword `midpoint_epoch` is renamed to `epoch_time`. [#9455]

astropy.uncertainty

- `Distribution` was rewritten such that it deals better with subclasses. As a result, Quantity distributions now behave correctly with `to` methods yielding new distributions of the kind expected for the starting distribution, and `to_value` yielding `NdarrayDistribution` instances. [#9442]

astropy.units

- For consistency with `ndarray`, scalar `Quantity.value` will now return a numpy scalar rather than a python one. This should help keep track of precision better, but may lead to unexpected results for the rare cases where numpy scalars behave differently than python ones (e.g., taking the square root of a negative number). [#8876]
- Removed the `magnitude_zero_points` module, which was deprecated in favour of `astropy.units.photometric` since 3.1. [#9353]
- `EquivalentUnitsList` now has a `_repr_html_` method to output a HTML table on a call to `find_equivalent_units` in Jupyter notebooks. [#9495]

astropy.utils

- `download_file` and related functions now accept a list of fallback sources, and they are able to update the cache at the user's request. [#9182]
- Allow `astropy.utils.console.ProgressBarOrSpinner.map` and `.map_unordered` to take an argument `multiprocessing_start_method` to control how subprocesses are started; the different methods (`fork`, `spawn`, and `forkserver`) have different implications in terms of security, efficiency, and behavioural anomalies. The option is useful in particular for cross-platform testing because Windows supports only `spawn` while Linux defaults to `fork`. [#9182]
- All operations that act on the astropy download cache now take an argument `pkgname` that allows one to specify which package's cache to use. [#8237, #9182]
- Removed deprecated `funcsigs` and `futures` from

`astropy.utils.compat`. [#8909]

- Removed the deprecated `astropy.utils.compat.numpy` module. [#8910]
- Deprecated `InheritDocstrings` as it is natively supported by Sphinx 1.7 or higher. [#8881]
- Deprecated `astropy.utils.timer` module, which has been moved to `astroquery.utils.timer` and will be part of `astroquery` 0.4.0. [#9038]
- Deprecated `astropy.utils.misc.set_locale` function, as it is meant for internal use only. [#9471]
- The implementation of `data_info.DataInfo` has changed (for a considerable performance boost). Generally, this should not affect simple subclasses, but because the class now uses `__slots__` any attributes on the class have to be explicitly given a slot. [#8998]
- `IERS` tables now use `nan` to mark missing values (rather than `1e20`). [#9226]

astropy.visualization

- The default `clip` value is now `False` in `ImageNormalize`. [#9478]
- The default `clip` value is now `False` in `simple_norm`. [#9698]
- Infinite values are now excluded when calculating limits in `ManualInterval` and `MinMaxInterval`. They were already excluded in all other interval classes. [#9480]

Bug Fixes

astropy.convolution

- Fixed `nan_treatment='interpolate'` option to `convolve_fft` to properly take into account `fill_value`. [#8122]

astropy.coordinates

- The `QuantityAttribute` class now supports a None default value if a unit is specified. [#9345]
- When `Representation` classes with the same name are defined, this no longer leads to a `ValueError`, but instead to a warning and the removal of both from the name registry (i.e., one either has to use the class itself to set, e.g., `representation_type`, or refer to the class by its fully qualified name). [#8561]

astropy.io.fits

- Implemented skip (after warning) of header cards with reserved keywords in `table_to_hdu`. [#9390]
- Add `AstropyDeprecationWarning` to `read_table_fits` when `hdu=` is selected, but does not match single present table HDU. [#9512]

astropy.io.votable

- Address issue #8995 by ignoring BINARY2 null mask bits for string values on parsing a VOTable. In this way, the reader should never create masked values for string types. [#9057]
- Corrected a spurious warning issued for the `value` attribute of the `<OPTION>` element in VOTable, as well as a test that erroneously treated the warning as acceptable. [#9470]

astropy.nddata

- `Cutout2D` will now get the WCS from its first argument if that argument has with WCS property. [#9492]
- `overlap_slices` will now raise a `ValueError` if the input position contains any non-finite values (e.g. NaN or inf). [#9648]

astropy.stats

- Fixed a bug where `bayesian_blocks` returned a single edge. [#8560]
- Fixed input data type validation for `bayesian_blocks` to work int arrays. [#9513]

astropy.table

- Fix bug where adding a column consisting of a list of masked arrays was dropping the masks. [#9048]
- `Quantity` columns with custom units can now round-trip via FITS tables, as long as the custom unit is enabled during reading (otherwise, the unit will become an `UnrecognizedUnit`). [#9015]
- Fix bug where string values could be truncated when inserting into a `Column` or `MaskedColumn`, or when adding or inserting a row containing string values. [#9559]

astropy.time

- Fix bug when `Time` object is created with only masked elements. [#9624]
- Fix inaccuracy when converting between TimeDelta and datetime.timedelta. [#9679]

astropy.units

- Ensure that output from test functions of and comparisons between quantities can be stored into pre-allocated output arrays (using `out=array`) [#9273]

astropy.utils

- For the default `IERS_Auto` table, which combines IERS A and B values, the IERS nutation parameters “dX_2000A” and “dY_2000A” are now also taken from the actual IERS B file rather than from the B values stored in the IERS A file. Any differences should be negligible for any practical application, but this may help exactly reproducing results. [#9237]

astropy.visualization

- Calling `WCSAxes.set_axis_off()` now correctly turns off drawing the Axes. [#9411]
- Fix incorrect transformation behavior in `WCSAxes.plot_coord` and correctly handle when input coordinates are not already in spherical representations. [#8927]
- Fixed `ImageNormalize` so that when it is intialized without `data` it will still use the input `interval` class. [#9698]
- Fixed `ImageNormalize` to handle input data with non-finite values. [#9698]

astropy.wcs

- Fix incorrect value returned by `wcsapi.HighLevelWCSWrapper.axis_correlation_matrix`. [#9554]
- Fix NaN-masking of world coordinates when some but not all of the coordinates were flagged as invalid by WCSLIB. This occurred for example with WCS with >2 dimensions where two of the dimensions were celestial coordinates and pixel coordinates ouside of the ‘sky’ were converted to world coordinates - previously all world coordinates were masked even if uncorrelated with the celestial axes, but this is no longer the case. [#9688]
- The default WCS to celestial frame mapping for world coordinate systems that specify `TLON` and `TLAT` coordinates will now return an ITRS frame with the representation class set to `SphericalRepresentation`. This fixes a bug that caused `WCS.pixel_to_world` to raise an error for such world coordinate systems. [#9609]
- `FITSWCSAPIMixin` now returns tuples not lists from `pixel_to_world` and `world_to_pixel`. [#9678]

Other Changes and Additions

- Versions of Python <3.6 are no longer supported. [#8955]
- Matplotlib 2.1 and later is now required. [#8787]
- Versions of Numpy <1.16 are no longer supported. [#9292]
- Updated the bundled CFITSIO library to 3.470. See `cextern/cfitsio/docs/changes.txt` for additional information. [#9233]
- The bundled ERFA was updated to version 1.7.0. This is based on SOFA 20190722. This includes a fix to avoid precision loss for negative JDs, and also includes additional routines to allow updates to the leap-second table. [#9323, #9734]
- The default server for the IERS data files has been updated to reflect long-term downtime of the canonical USNO server. [#9487, #9508]

3.2.3 (2019-10-27)

Other Changes and Additions

- Updated IERS A URLs due to USNO prolonged maintenance. [#9443]

3.2.2 (2019-10-07)

Bug fixes

astropy.convolution

- Fixed a bug in `discretize_oversample_1D/2D()` from `astropy.convolution.utils`, which might occasionally introduce unexpected oversampling grid dimensions due to a numerical precision issue. [#9293]
- Fixed a bug [#9168] where having a kernel defined using unitless astropy quantity objects would result in a crash [#9300]

astropy.coordinates

- Fix concatenation of representations for cases where the units were different. [#8877]
- Check for NaN values in catalog and match coordinates before building and querying the `KDTree` for coordinate matching. [#9007]
- Fix sky coordinate matching when a dimensionless distance is provided. [#9008]
- Raise a faster and more meaningful error message when differential data units are not compatible with a containing representation’s units. [#9064]
- Changed the timescale in ICRS to CIRS from ‘tdb’ to ‘tt’ conversion and vice-versa, as the erfa function that gets called in the process, pnm06a accepts

time in TT. [#9079]

astropy.io.ascii

- Fixed the fast reader when used in parallel and with the multiprocessing 'spawn' method (which is the default on MacOS X with Python 3.8 and later), and enable parallel fast reader on Windows. [#8853]

astropy.io.fits

- Fixes bug where an invalid TRPOS<n> keyword was being generated for FITS time column when no location was available. [#8784]
- Fixed a wrong exception when converting a Table with a unit that is not FITS compliant and not convertible to a string using `format='fits'`. [#8906]
- Fixed an issue with A3DTABLE extension that could not be read. [#9012]
- Fixed the update of the header when creating GroupsHDU from data. [#9216]

astropy.nddata

- Fix to `add_array`, which now accepts `array_small` having dimensions equal to `array_large`, instead of only allowing smaller sizes of arrays. [#9118]

astropy.stats

- Fixed `median_absolute_deviation` for the case where `ignore_nan=True` and an input masked array contained both NaNs and infs. [#9307]

astropy.table

- Comparisons between `Column` instances and `Quantity` will now correctly take into account the unit (as was already the case for regular operations such as addition). [#8904]

astropy.time

- Allow `Time` to be initialized with an empty value for all formats. [#8854]
- Fixed a troubling bug in which `Time` could loose precision, with deviations of 300 ns. [#9328]

astropy.timeseries

- Fixed handling of `Quantity` input data for all methods of `LombScarge.false_alarm_probabilty`. [#9246]

astropy.units

- Allow conversion of `Column` with logarithmic units to a suitable `Quantity` subclass if `subok=True`. [#9188]
- Ensured that we simplify powers to smaller denominators if that is consistent within rounding precision. [#9267]
- Ensured that the powers shown in a unit's repr are always correct, not oversimplified. [#9267]

astropy.utils

- Fixed `find_api_page` access by using custom request headers and HTTPS when version is specified. [#9032]
- Make `download_file` (and by extension `get_readable_fileobj` and others) check the size of downloaded files against the size claimed by the server. [#9302]
- Fix `find_current_module` so that it works properly if astropy is being used inside a bundle such as that produced by PyInstaller. [#8845]
- Fix path to renamed classes, which previously included duplicate path/module information under certain circumstances. [#8845]

astropy.visualization

- Silence numpy runtime warnings in `WCSAxes` when drawing grids. [#8882]

astropy.wcs

- Fixed equality test between `cunit` where the first element was equal but the following elements differed. [#9154]
- Fixed a crash while loading a WCS from headers containing duplicate SIP keywords. [#8893]
- Fixed a possible buffer overflow when using too large negative indices for `cunit` or `ctype` [#9151]
- Fixed reference counting in `WCSBase.__init__` [#9166]
- Fix `SlicedLowLevelWCS` `world_to_pixel_values` and `pixel_to_world_values` when inputs need broadcasting to the same shape. (i.e. when one input is sliced out) [#9250]
- Fixed a bug that caused `WCS.array_shape`, `WCS.pixel_shape` and `WCS.pixel_bounds` to be incorrect after using `WCS.sub`. [#9095]

Other Changes and Additions

- Fixed a bug that caused files outside of the astropy module directory to be included as package data, resulting in some cases in errors when doing repeated builds. [#9039]

3.2.1 (2019-06-14)

Bug fixes

astropy.io.fits

- Avoid reporting a warning with `BinTableHDU.from_columns` with keywords that are not provided by the user. [#8838]
- Fix `Header.fromfile` to work on FITS files. [#8713]
- Fix reading of empty `BinTableHDU` when stored in a gzip-compressed file. [#8848]

astropy.table

- Fix a problem where mask was dropped when creating a `MaskedColumn` from a list of `MaskedArray` objects. [#8826]

astropy.wcs

- Added `None` to be displayed as a `world_axis_physical_types` in the `WCS` repr, as `None` values are now supported in `APE14`. [#8811]

3.2 (2019-06-10)

New Features

astropy.constants

- Add CODATA 2018 constants but not make them default because the redefinition of SI units that will follow has not been implemented yet. [#8595]

astropy.coordinates

- New `BarycentricMeanEcliptic`, `HeliocentricTrueEcliptic` and `GeocentricTrueEcliptic` frames. The ecliptic frames are no longer considered experimental. [#8394]
- The default time scale for epochs like ‘J2000’ or ‘B1975’ is now “tt”, which is the correct one for ‘J2000’ and avoids leap-second warnings for epochs in the far future or past. [#8600]

astropy.extern

- Bundled `six` now emits `AstropyDeprecationWarning`. It will be removed in 4.0. [#8323]

astropy.io.ascii

- IPAC tables now output data types of `float` instead of `double`, or `int` instead of `long`, based on the column `dtype.itemsize`. [#8216]
- Update handling of MaskedColumn columns when using the 'data_mask' serialization method. This can make writing ECSV significantly faster if the data do not actually have any masked values. [#8447]
- Fixed a bug that caused newlines to be incorrect when writing out ASCII tables on Windows (they were `\r\r\n` instead of `\r\n`). [#8659]

astropy.io.misc

- Implement serialization of `TimeDelta` in ASDF. [#8285]
- Implement serialization of `EarthLocation` in ASDF. [#8286]
- Implement serialization of `SkyCoord` in ASDF. [#8284]
- Support serialization of Astropy tables with mixin columns in ASDF. [#8337]
- No warnings when reading HDF5 files with only one table and no `path=` argument [#8483]
- The HDF5 writer will now create a default table instead of raising an exception when `path=` is not specified and when writing to empty/new HDF5 files. [#8553]

astropy.io.fits

- Optimize parsing of cards within the `Header` class. [#8428]
- Optimize the parsing of headers to get the structural keywords that are needed to find extensions. Thanks to this, getting a random HDU from a file with many extensions is much faster than before, in particular when the extension headers contain many keywords. [#8502]
- Change behavior of FITS undefined value in `Header` such that `None` is used in Python to represent FITS undefined when using dict interface. `Undefined` can also be assigned and is translated to `None`. Previously setting a header card value to `None` resulted in an empty string field rather than a FITS undefined value. [#8572]
- Allow `Header.fromstring` and `Card.fromstring` to accept `bytes`. [#8707]

astropy.io.registry

- Implement `Table` reader and writer for `ASDF`. [#8261]
- Implement `Table` reader and writer methods to wrap `pandas` I/O

methods for CSV, Fixed width format, HTML, and JSON. [#8381]
- Add `help()` and `list_formats()` methods to unified I/O `read` and `write` methods. For example `Table.read.help()` gives help on available `Table` read formats and `Table.read.help('fits')` gives detailed help on the arguments for reading FITS table file. [#8255]

astropy.table

- Initializing a table with `Table(rows=...)`, if the first item is an `OrderedDict`, now uses the column order of the first row. [#8587]
- Added new pprint_all() and pformat_all() methods to Table. These two new methods print the entire table by default. [#8577]
- Removed restriction of initializing a Table from a dict with copy=False. [#8541]
- Improved speed of table row access by a factor of about 2-3. Improved speed of Table len() by a factor of around 3-10 (depending on the number of columns). [#8494]
- Improved the Table - pandas `DataFrame` interface (`to_pandas()` and `from_pandas()`). Mixin columns like `Time` and `Quantity` can now be converted to pandas by flattening the columns as necessary to plain columns. `Time` and `TimeDelta` columns get converted to corresponding pandas date or time delta types. The `DataFrame` index is now handled in the conversion methods. [#8247]
- Added `rename_columns` method to rename multiple columns in one call. [#5159, #8070]
- Improved Table performance by reducing unnecessary calls to copy and deepcopy, especially as related to the table and column `meta` attributes. Changed the behavior when slicing a table (either in rows or with a list of column names) so now the sliced output gets a light (key-only) copy of `meta` instead of a deepcopy. Changed the `Table.meta` class-level descriptor so that assigning directly to `meta`, e.g. `tbl.meta = new_meta` no longer does a deepcopy and instead just directly assigns the `new_meta` object reference. Changed Table initialization so that input `meta` is copied only if `copy=True`. [#8404]
- Improved Table slicing performance with internal implementation changes related to column attribute access and certain input validation. [#8493]
- Added `reverse` argument to the `sort` and `argsort` methods to allow sorting in reverse order. [#8528]
- Improved `Table.sort()` performance by removing `self[keys]` from code which is creating deep copies of `meta` attribute and adding a new keyword `names` in `get_index()` to get index by using a list or tuple containing names of columns. [#8570]

- Expose `represent_mixins_as_columns` as a public function in the `astropy.table` subpackage. This previously-private function in the `table.serialize` module is used to represent mixin columns in a Table as one or more plain Column objects. [#7729]

astropy.timeseries

- Added a new astropy.timeseries sub-package to represent and manipulate sampled and binned time series. [#8540]
- The `BoxLeastSquares` and `LombScargle` classes have been moved to `astropy.timeseries.periodograms` from `astropy.stats`. [#8591]
- Added the ability to provide absolute `Time` objects to the `BoxLeastSquares` and `LombScargle` periodogram classes. [#8599]
- Added model inspection methods (`model_parameters()`, `design_matrix()`, and `offset()`) to `astropy.timeseries.LombScargle` class [#8397].

astropy.units

- `Quantity` overrides of `ndarray` methods such as `sum`, `min`, `max`, which are implemented via reductions, have been removed since they are dealt with in `Quantity.__array_ufunc__`. This should not affect subclasses, but they may consider doing similarly. [#8316] Note that this does not include methods that use more complicated python code such as `mean`, `std` and `var`. [#8370]

astropy.visualization

- Added `CompositeStretch`, which inherits from `CompositeTransform` and also `BaseStretch` so that it can be used with `ImageNormalize`. [#8564]
- Added a `log_a` argument to the `simple_norm` method. Similar to the exposing of the `asinh_a` argument for `AsinhStretch`, the new `log_a` argument is now exposed for `LogStretch`. [#8436]

astropy.wcs

- WCSLIB was updated to v 6.2. This adds support for time-related WCS keywords (WCS Paper VII). FITS headers containing `Time` axis are parsed and the axis is included in the WCS object. [#8592]
- The `OBSGEO` attribute as expanded to 6 members - `XYZLBH`. [#8592]
- Added a new class `SlicedLowLevelWCS` in `astropy.wcs.wcsapi` that can be used to slice any WCS that conforms to the `BaseLowLevelWCS`

API. [#8546]
- Updated implementation of `WCS.__getitem__` and `WCS.slice` to now return a `SlicedLowLevelWCS` rather than raising an error when reducing the dimensionality of the WCS. [#8546]

API Changes

astropy.coordinates

- `QuantityAttribute` no longer has a default value for `default`. The previous value of None was misleading as it always was an error. [#8450]
- The default J2000 has been changed to use be January 1, 2000 12:00 TT instead of UTC. This is more in line with convention. [#8594]

astropy.io.ascii

- IPAC tables now output data types of `float` instead of `double`, or `int` instead of `long`, based on the column `dtype.itemsize`. [#8216]

astropy.io.misc

- Unit equivalencies can now be serialized to ASDF. [#8252]

astropy.modeling

- Composition of model classes is deprecated and will be removed in 4.0. Composition of model instances remain unaffected. [#8234, #8408]

astropy.stats

- The `BoxLeastSquares` and `LombScargle` classes have been moved to the `astropy.timeseries.periodograms` module and will now emit a deprecation warning when imported from `astropy.stats`. [#8591]

astropy.table

- Converting an empty table to an array using `as_array` method now returns an empty array instead of `None`. [#8647]
- Changed the behavior when slicing a table (either in rows or with a list of column names) so now the sliced output gets a light (key-only) copy of `meta` instead of a deepcopy. Changed the `Table.meta` class-level descriptor so that assigning directly to `meta`, e.g. `tbl.meta = new_meta` no longer does a deepcopy and instead just directly assigns the `new_meta` object reference. Changed Table initialization so that input `meta` is copied only if `copy=True`. [#8404]

- Added a keyword `names` in `Table.as_array()`. If provided this specifies a list of column names to include for the returned structured array. [#8532]

astropy.tests

- Removed `pytest_plugins` as they are completely broken for `pytest>=4`. [#7786]
- Removed the `astropy.tests.plugins.config` plugin and removed the `--astropy-config-dir` and `--astropy-cache-dir` options from testing. Please use caching functionality that is natively in `pytest`. [#7787, #8489]

astropy.time

- The default time scale for epochs like 'J2000' or 'B1975' is now "tt", which is the correct one for 'J2000' and avoids leap-second warnings for epochs in the far future or past. [#8600]

astropy.units

- Unit equivalencies can now be introspected. [#8252]

astropy.wcs

- The `world_to_pixel`, `world_to_array_index*`, `pixel_to_world*` and `array_index_to_world*` methods now all consistently return scalars, arrays, or objects not wrapped in a one-element tuple/list when only one scalar, array, or object (as was previously already the case for `WCS.pixel_to_world` and `WCS.array_index_to_world`). [#8663]

astropy.utils

- It is now possible to control the number of cores used by `ProgressBar.map` by passing a positive integer as the `multiprocess` keyword argument. Use `True` to use all cores. [#8083]

Bug Fixes

astropy.coordinates

- `BarycentricTrueEcliptic`, `HeliocentricTrueEcliptic` and `GeocentricTrueEcliptic` now use the correct transformation (including nutation), whereas the new `*MeanEcliptic` classes use the nutation-free

transformation. [#8394]
- Representations with `float32` coordinates can now be transformed, although the output will always be `float64`. [#8759]
- Fixed bug that prevented using differentials with HCRS<->ICRS transformations. [#8794]

astropy.io.ascii

- Fixed a bug where an exception was raised when writing a table which includes mixin columns (e.g. a Quantity column) and the output format was specified using the `formats` keyword. [#8681]

astropy.io.misc

- Fixed bug in ASDF tag that inadvertently introduced dependency on `pytest`. [#8456]

astropy.modeling

- Fixed slowness for certain compound models consisting of large numbers of multi-input models [#8338, #8349]
- Fixed bugs in fitting of compound models with units. [#8369]

astropy.nddata

- Fixed bug in reading multi-extension FITS files written by earlier versions of `CCDData`. [#8534]
- Fixed two errors in the way `CCDData` handles FITS files with WCS in the header. Some of the WCS keywords that should have been removed from the header were not, potentially leading to FITS files with inconsistent WCS. [#8602]

astropy.table

- Fixed a bug when initializing from an empty list: `Table([])` no longer results in a crash. [#8647]
- Fixed a bug when initializing from an existing `Table`. In this case the input `meta` argument was being ignored. Now the input `meta`, if supplied, will be used as the `meta` for the new `Table`. [#8404]
- Fix the conversion of bytes values to Python `str` with `Table.tolist`. [#8739]

astropy.time

- Fixed a number of issues to ensure a consistent output type resulting from multiplication or division involving a `TimeDelta` instance. The output is

now always a `TimeDelta` if the result is a time unit (like u.s or u.d), otherwise it will be a `Quantity`. [#8356]
- Multiplication between two `TimeDelta` instances is now possible, resulting in a `Quantity` with units of time squared (division already correctly resulted in a dimensionless `Quantity`). [#8356]
- Like for comparisons, addition, and subtraction of `Time` instances with with non-time instances, multiplication and division of `TimeDelta` instances with incompatible other instances no longer immediately raise an `UnitsError` or `TypeError` (depending on the other instance), but rather go through the regular Python mechanism of `TimeDelta` returning `NotImplemented` (which will lead to a regular `TypeError` unless the other instance can handle `TimeDelta`). [#8356]
- Corrected small rounding errors that could cause the `jd2` values in `Time` to fall outside the range of -0.5 to 0.5. [#8763]

astropy.units

- Added a `Quantity.to_string` method to add flexibility to the string formatting of quantities. It produces unadorned or LaTeX strings, and accepts two different sets of delimiters in the latter case: `inline` and `display`. [#8313]
- Ensure classes that mimic quantities by having a `unit` attribute and/or `to` and `to_value` methods can be properly used to initialize `Quantity` or set `Quantity` instance items. [#8535]
- Add support for `<<` to create logarithmic units. [#8290]
- Add support for the `clip` ufunc, which in numpy 1.17 is used to implement `np.clip`. As part of that, remove the `Quantity.clip` method under numpy 1.17. [#8747]
- Fix parsing of numerical powers in FITS-compatible units. [#8251]

astropy.wcs

- Added a `PyUnitListProxy_richcmp` method in `UnitListProxy` class to enable `WCS.wcs.cunit` equality testing. It helps to check whether the two instances of `WCS.wcs.cunit` are equal or not by comparing the data members of `UnitListProxy` class [#8480]
- Fixed `SlicedLowLevelWCS` when `array_shape` is `None`. [#8649]
- Do not attempt to delete repeated distortion keywords multiple times when loading distortions with `_read_distortion_kw` and `_read_det2im_kw`. [#8777]

Other Changes and Additions

- Update bundled expat to 2.2.6. [#8343]
- Added instructions for uploading releases to Zenodo. [#8395]
- The bug fixes to the behaviour of `TimeDelta` for multiplcation and division, which ensure that the output is now always a `TimeDelta` if the result is a time unit (like u.s or u.d) and otherwise a `Quantity`, imply that sometimes the output type will be different than it was before. [#8356]
- For types unrecognized by `TimeDelta`, multiplication and division now will consistently return a `TypeError` if the other instance cannot handle `TimeDelta` (rather than `UnitsError` or `TypeError` depending on presumed abilities of the other instance). [#8356]
- Multiplication between two `TimeDelta` instances will no longer result in an `OperandTypeError`, but rather result in a `Quantity` with units of time squared (division already correctly resulted in a dimensionless `Quantity`). [#8356]
- Made running the tests insensitive to local user configuration when running the tests in parallel mode or directly with pytest. [#8727]
- Added a narrative style guide to the documentation for contributor reference. [#8588]
- Ensure we call numpy equality functions in a way that reduces the number of `DeprecationWarning`. [#8755]

Installation

- We now require setuptools 30.3.0 or later to install the core astropy package. [#8240]
- We now define groups of dependencies that can be installed with pip, e.g. `pip install astropy[all]` (to install all optional dependencies). [#8198]

3.1.2 (2019-02-23)

Bug fixes

astropy.coordinates

- Convert the default of `QuantityAttribute`, thereby catching the error case case of it being set to None at attribute creation, and giving a more useful error message in the process. [#8300]

astropy.cosmology

- Fix elliptic analytical solution for comoving distance. Only relevant for non-flat cosmologies without radiation and `Om0` > `Ode0`. [#8391]

astropy.modeling

- Fixed slowness for certain compound models consisting of large numbers of multi-input models [#8338, #8349]

astropy.visualization.wcsaxes

- Fix a bug that caused an error when passing an array with all values the same to contour or contourf. [#8321]
- Fix a bug that caused contour and contourf to return None instead of the contour set. [#8321]

3.1.1 (2018-12-31)

Bug fixes

astropy.io.ascii

- Fix error when writing out empty table. [#8279]

astropy.io.fits

- `fitsdiff --ignore-hdus` now prints input filenames in the diff report instead of `<HDUList object at 0x1150f9778>`. [#8295]

astropy.units

- Ensure correctness of units when raising to a negative power. [#8263]
- Fix `with_H0` equivalency to use the correct direction of conversion. [#8292]

3.1 (2018-12-06)

New Features

astropy.convolution

- `convolve` now accepts any array-like input, not just `numpy.ndarray` or lists. [#7303]
- `convolve` Now raises AstropyUserWarning if nan_treatment='interpolate' and preserve_nan=False and NaN values are present post convolution. [#8088]

astropy.coordinates

- The `SkyCoord.from_name` constructor now has the ability to create coordinate objects by parsing object catalogue names that have embedded J-coordinates. [#7830]
- The new function `make_transform_graph_docs` can be used to create a docstring graph from a custom `TransformGraph` object. [#7135]
- `KDTree` for catalog matching is now built with sliding midpoint rule rather than standard. In code, this means setting `compact_nodes=False` and `balanced_tree=False` in `cKDTree`. The sliding midpoint rule is much more suitable for catalog matching, and results in 1000x speedup in some cases. [#7324]
- Additional information about a site loaded from the Astropy sites registry is now available in `EarthLocation.info.meta`. [#7857]
- Added a `concatenate_representations` function to combine coordinate representation data and any associated differentials. [#7922]
- `BaseCoordinateFrame` will now check for a method named `_astropy_repr_in_frame` when constructing the string forms of attributes. Allowing any class to control how `BaseCoordinateFrame` represents it when it is an attribute of a frame. [#7745]
- Some rarely-changed attributes of frame classes are now cached, resulting in speedups (up to 50% in some cases) when creating new scalar frame or `SkyCoord` objects. [#7949, #5952]
- Added a `directional_offset_by` method to `SkyCoord` that computes a new coordinate given a coordinate, position angle, and angular separation [#5727]

astropy.cosmology

- The default cosmology has been changed from `WMAP9` to `Planck15`. [#8123]
- Distance calculations with `LambaCDM` with no radiation (T_CMB0=0) are now 20x faster by using elliptic integrals for non-flat cases. [#7155]
- Distance calculations with `FlatLambaCDM` with no radiation (T_CMB0=0) are now 20x faster by using the hypergeometric function solution for this special case. [#7087]
- Age calculations with `FlatLambdaCDM` with no radiation (Tcmb0=0) are now 1000x faster by using analytic solutions instead of integrating. [#7117]

astropy.io.ascii

- Latex reader now ignores `\toprule`, `\midrule`, and `\bottomrule` commands. [#7349]

- Added the RST (Restructured-text) table format and the fast version of the RDB reader to the set of formats that are guessed by default. [#5578]
- The read trace (used primarily for debugging) now includes guess argument sets that were skipped entirely e.g. for not supporting user-supplied kwargs. All guesses thus removed from `filtered_guess_kwargs` are now listed as “Disabled” at the beginning of the trace. [#5578]
- Emit a warning when reading an ECSV file without specifying the `format` and without PyYAML installed. Previously this silently fell through to parsing as a basic format file and the file metadata was lost. [#7580]
- Optionally allow writing masked columns to ECSV with the mask explicitly specified as a separate column instead of marking masked elements with “” (empty string). This allows handling the case of a masked string column with “” data rows. [#7481]

astropy.io.misc

- Added support for saving all representation classes and many coordinate frames to the asdf format. [#7079]
- Added support for saving models with units to the asdf format. [#7237]
- Added a new `character_as_bytes` keyword to the HDF5 Table reading function to control whether byte string columns in the HDF5 file are left as bytes or converted to unicode. The default is to read as bytes (`character_as_bytes=True`). [#7024, #8017]

astropy.io.fits

- `HDUList.pop()` now accepts string and tuple extension name specifications. [#7236]
- Add an `ignore_hdus` keyword to `FITSDiff` to allow ignoring HDUs by NAME when diffing two FITS files [#7538]
- Optionally allow writing masked columns to FITS with the mask explicitly specified as a separate column instead of using the FITS standard of certain embedded null values (`NaN` for float, `TNULL` for integers). This can be used to work around limitations in the FITS standard. [#7481]
- All time coordinates can now be written to and read from FITS binary tables, including those with vectorized locations. [#7430]
- The `fitsheader` command line tool now supports a `dfits+fitsort` mode, and the dotted notation for keywords (e.g. `ESO.INS.ID`). [#7240]
- Fall back to reading arrays using mode=’denywrite’ if mode=’readonly’ fails when using memory-mapping. This solves cases on some platforms when the available address space was less than the file size (even when using memory mapping). [#7926]

astropy.modeling

- Add a `Multiply` model which preserves unit through evaluate, unlike `Scale` which is dimensionless. [#7210]
- Add a `uses_quantity` property to `Model` which allows introspection of if the `Model` can accept `Quantity` objects. [#7417]
- Add a `separability_matrix` function which returns the correlation matrix of inputs and outputs. [#7803]
- Fixed compatibility of `JointFitter` with the latest version of Numpy. [#7984]
- Add `prior` and `posterior` constraints to modeling parameters. These are not used by any current fitters, but are provided to allow user code to experiment with Bayesian fitters. [#7558]

astropy.nddata

- `NDUncertainty` objects now have a `quantity` attribute for simple conversion to quantities. [#7704]
- Add a `bitmask` module that provides functions for manipulating bitmasks and data quality (DQ) arrays. [#7944]

astropy.stats

- Add an `astropy.stats.bls` module with an implementation of the “box least squares” periodogram that is commonly used for discovering transiting exoplanets and eclipsing binaries. [#7391]

astropy.table

- Added support for full use of `Time` mixin column for join, hstack, and vstack table operations. [#6888]
- Added a new table index engine, `SCEngine`, based on the Sorted Containers package. [#7574]
- Add a new keyword argument `serialize_method` to `Table.write` to control how `Time` and `MaskedColumn` columns are written. [#7481]
- Allow mixin columns to be used in table `group` and `unique` functions. This applies to both the key columns and the other data columns. [#7712]
- Added support for stacking `Column`, mixin column (e.g. `Quantity`, `Time`) or column-like objects. [#7674]
- Added support for inserting a row into a Table that has `Time` or `TimeDelta` column(s). [#7897]

astropy.tests

- Added an option `--readonly` to the test command to change the permissions on the temporary installation location to read-only. [#7598]

astropy.time

- Allow array-valued `Time` object to be modified in place. [#6028]
- Added support for missing values (masking) to the `Time` class. [#6028]
- Added supper for a 'local' time scale (for free-running clocks, etc.), and round-tripping to the corresponding FITS time scale. [#7122]
- Added **`datetime.timedelta`** format class for `TimeDelta`. [#7441]
- Added `strftime` and `strptime` methods to `Time` class. These methods are similar to those in the Python standard library **`time`** package and provide flexible input and output formatting. [#7323]
- Added `datetime64` format to the `Time` class to support working with `numpy.datetime64` dtype arrays. [#7361]
- Add fractional second support for `strftime` and `strptime` methods of `Time` class. [#7705]
- Added an `insert` method to allow inserting one or more values into a `Time` or `TimeDelta` object. [#7897]
- Remove timescale from string version of FITS format time string. The timescale is not part of the FITS standard and should not be included. This change may cause some compatibility issues for code that relies on round-tripping a FITS format string with a timescale. Strings generated from previous versions of this package are still understood but a DeprecationWarning will be issued. [#7870]

astropy.uncertainty

- This sub-package was added as a "preview" (i.e. API unstable), containing the `Distribution` class and associated convenience functions. [#6945]

astropy.units

- Add complex numbers support for `Quantity._repr_latex_`. [#7676]
- Add `thermodynamic_temperature` equivalency to convert between Jy/sr and "thermodynamic temperature" for cosmology. [#7054]
- Add millibar unit. [#7863]
- Add maggy and nanomaggy unit, as well as associated `zero_point_flux` equivalency. [#7891]
- `AB` and `ST` are now enabled by default, and have alternate names `ABflux` and `STflux`. [#7891]
- Added `littleh` unit and associated `with_H0` equivalency. [#7970]

astropy.visualization

- Added `imshow_norm` function, which combines imshow and creation of a

`ImageNormalize` object. [#7785]

astropy.visualization.wcsaxes

- Add support for setting `set_separator(None)` in WCSAxes to use default separators. [#7570]
- Added two keyword argument options to `CoordinateHelper.set_format_unit`: `decimal` can be used to specify whether to use decimal formatting for the labels (by default this is False for degrees and hours and True otherwise), and `show_decimal_unit` can be used to determine whether the units should be shown for decimal labels. [#7318]
- Added documentation for `transform=` and `coord_meta=`. [#7698]
- Allow `coord_meta=` to optionally include `format_unit=`. [#7848]
- Add support for more rcParams related to the grid, ticks, and labels, and should work with most built-in Matplotlib styles. [#7961]
- Improved rendering of outward-facing ticks. [#7961]
- Add support for `tick_params` (which is a standard Matplotlib function/method) on both the `WCSAxes` class and the individual `CoordinateHelper` classes. Note that this is provided for compatibility with Matplotlib syntax users may be familiar with, but it is not the preferred way to change settings. Instead, methods such as `set_ticks` should be preferred. [#7969]
- Moved the argument `exclude_overlapping` from `set_ticks` to `set_ticklabel`. [#7969]
- Added a `pad=` argument to `set_ticklabel` to provide a way to control the padding between ticks and tick labels. [#7969]
- Added support for setting the tick direction in `set_ticks` using the `direction=` keyword argument. [#7969]

astropy.wcs

- Map ITRS frames to terrestrial WCS coordinates. This will make it possible to use WCSAxes to make figures that combine both celestial and terrestrial features. An example is plotting the coordinates of an astronomical transient over an all- sky satellite image to illustrate the position relative to the Earth at the time of the event. The ITRS frame is identified with WCSs that use the `TLON-` and `TLAT-` coordinate types. There are several examples of WCSs where this syntax is used to describe terrestrial coordinate systems: Section 7.4.1 of WCS in FITS “Paper II” and the WCSTools documentation. [#6990]
- Added the abstract base class for the low-level WCS API described in APE 14 (https://doi.org/10.5281/zenodo.1188875). [#7325]

- Add `WCS.footprint_contains()` function to check if the WCS footprint contains a given sky coordinate. [#7273]
- Added the abstract base class for the high-level WCS API described in APE 14 (https://doi.org/10.5281/zenodo.1188875). [#7325]
- Added the high-level wrapper class for low-level WCS objects as described in APE 14 (https://doi.org/10.5281/zenodo.1188875). [#7326]
- Added a new property `WCS.has_distortion`. [#7326]
- Deprecated `_naxis1` and `_naxis2` in favor of `pixel_shape`. [#7973]
- Added compatibility to wcslib version 6. [#8093]

API Changes

astropy.convolution

- `kernel` can now be a tuple. [#7561]
- Not technically an API changes, however, the doc string indicated that `boundary=None` was the default when actually it is `boundary='fill'`. The doc string has been corrected, however, someone may interpret this as an API change not realising that nothing has actually changed. [#7293]
- `interpolate_replace_nans()` can no longer accept the keyword argument `preserve_nan`. It is explicitly set to `False`. [#8088]

astropy.coordinates

- Fixed `astropy.coordinates.concatenate` to include velocity data in the concatenation. [#7922]
- Changed the name of the single argument to `Frame.realize_frame()` from the (incorrect) `representation_type` to `data`. [#7923]
- Negative parallaxes passed to `Distance()` now raise an error by default (`allow_negative=False`), or are converted to NaN values with a warning (`allow_negative=True`). [#7988]
- Negating a `SphericalRepresentation` object now changes the angular coordinates (by rotating 180º) instead of negating the distance. [#7988]
- Creation of new frames now generally creates copies of frame attributes, rather than inconsistently either copying or making references. [#8204]
- The frame class method `is_equivalent_frame` now checks for equality of components to determine if a frame is the same when it has frame attributes that are representations, rather than checking if they are the same object. [#8218]

astropy.io.ascii

- If a fast reader is explicitly selected (e.g. `fast_reader='force'`) and

options which are incompatible with the fast reader are provided (e.g. `quotechar='##'`) then now a `ParameterError` exception will be raised. [#5578]
- The fast readers will now raise `InconsistentTableError` instead of `CParserError` if the number of data and header columns do not match. [#5578]
- Changed a number of `ValueError` exceptions to `InconsistentTableError` in places where the exception is related to parsing a table which is inconsistent with the specified table format. Note that `InconsistentTableError` inherits from `ValueError` so no user code changes are required. [#7425]

astropy.io.fits

- The `fits.table_to_hdu()` function will translate any column `format` attributes to a TDISPn format string, if possible, and store it as a TDISPn keyword in the `HDU` header. [#7226]

astropy.modeling

- Change the order of the return values from `FittingWithOutlierRemoval`, such that `fitted_model` comes first, for consistency with other fitters. For the second value, return only a boolean outlier `mask`, instead of the previous `MaskedArray` (which included a copy of the input data that was both redundant and inadvertently corrupted at masked points). Return a consistent type for the second value when `niter=0`. [#7407]
- Set the minimum value for the `bolometric_flux` parameter of the `BlackBody1D` model to zero. [#7045]

astropy.nddata

- Add two new uncertainty classes, `astropy.nddata.VarianceUncertainty` and `astropy.nddata.InverseVariance`. [#6971]

astropy.stats

- String values can now be used for the `cenfunc` and `stdfunc` keywords in the `SigmaClip` class and `sigma_clip` and `sigma_clipped_stats` functions. [#7478]
- The `SigmaClip` class and `sigma_clip` and `sigma_clipped_stats` functions now have a `masked` keyword, which can be used to return either a masked array (default) or an ndarray with the min/max values. [#7478]

- The `iters` keyword has been renamed (and deprecated) to `maxiters` in the `SigmaClip` class and `sigma_clip` and `sigma_clipped_stats` functions. [#7478]

astropy.table

- `Table.read()` on a FITS binary table file will convert any TDISPn header keywords to a Python formatting string when possible, and store it in the column `format` attribute. [#7226]
- No values provided to stack will now raise `ValueError` rather than `TypeError`. [#7674]

astropy.tests

- `from astropy.tests.helper import *` no longer includes `quantity_allclose`. However, `from astropy.tests.helper import quantity_allclose` would still work. [#7381]
- `warnings_to_ignore_by_pyver` option in `enable_deprecations_as_exceptions()` now takes `None` as key. Any deprecation message that is mapped to `None` will be ignored regardless of the Python version. [#7790]

astropy.time

- Added the ability to use `local` as time scale in `Time` and `TimeDelta`. [#6487]
- Comparisons, addition, and subtraction of `Time` instances with non-time instances will now return `NotImplemented` rather than raise the `Time`-specific `OperandTypeError`. This will generally lead to a regular `TypeError`. As a result, `OperandTypeError` now only occurs if the operation is between `Time` instances of incompatible type or scale. [#7584]

astropy.units

- In `UnitBase.compose()`, if a sequence (list|tuple) is passed in to `units`, the default for `include_prefix_units` is set to **`True`**, so that no units get ignored. [#6957]
- Negative parallaxes are now converted to NaN values when using the `parallax` equivalency. [#7988]

astropy.utils

- `InheritDocstrings` now also works on class properties. [#7166]
- `diff_values()`, `report_diff_values()`, and

`where_not_allclose()` utility functions are moved from `astropy.io.fits.diff`. [#7444]
- `invalidate_caches()` has been removed from the `astropy.utils.compat` namespace, use it directly from `importlib`. [#7872]

astropy.visualization

- In `ImageNormalize`, the default for `clip` is set to `True`. [#7800]
- Changed `AsymmetricPercentileInterval` and `MinMaxInterval` to ignore NaN values in arrays. [#7360]
- Automatically default to using `grid_type='contours'` in WCSAxes when using a custom `Transform` object if the transform has no inverse. [#7847]

Performance Improvements

- Reduced import time by more cautious use of the standard library. [#7647]

astropy.convolution

- Major performance overhaul to `convolve()`. [#7293]
- `convolve()`: Boundaries `fill`, `extend`, and `wrap` now use a single implementation that pads the image with the correct boundary values before convolving. The runtimes of these three were significantly skewed. They now have equivalent runtimes that are also faster than before due to performant contiguous memory access. However, this does increase the memory footprint as an entire new image array is required plus that needed for the padded region.[#7293]
- `convolve()`: Core computation ported from Cython to C. Several optimization techniques have been implemented to achieve performance gains, e.g. compiler hoisting, and vectorization, etc. Compiler optimization level `-O2` required for hoisting and `-O3` for vectorization. [#7293]
- `convolve()`: `nan_treatment=‘interpolate’` was slow to compute irrespective of whether any NaN values exist within the array. The input array is now checked for NaN values and interpolation is disabled if non are found. This is a significant performance boost for arrays without NaN values. [#7293]

astropy.coordinates

- Sped up creating SkyCoord objects by a factor of ~2 in some cases. [#7615]
- Sped up getting xyz vectors from `CartesianRepresentation` (which is used a lot internally). [#7638]

- Sped up transformations and some representation methods by replacing python code with (compiled) `erfa` ufuncs. [#7639]
- Sped up adding differential (velocity) data to representations by a factor of ~20, which improves the speed of frame and SkyCoord initialization. [#7924]
- Refactored `SkyCoord` initializer to improve performance and code clarity. [#7958]
- Sped up initialization of `Longitude` by ~40%. [#7616]

astropy.stats

- The `SigmaClip` class and `sigma_clip` and `sigma_clipped_stats` functions are now significantly faster. [#7478]
- A Cython implementation for **`astropy.stats.kuiper_two`** and a vectorized implementation for **`astropy.stats.kuiper_false_positive_probability`** have been added, speeding up both functions. [#8104]

astropy.units

- Sped up creating new composite units, and raising units to some power [#7549, #7649]
- Sped up Unit.to when target unit is the same as the original unit. [#7643]
- Lazy-load `scipy.special` to shorten `astropy.units` import time. [#7636]

astropy.visualization

- Significantly sped up drawing of contours in WCSAxes. [#7568]

Bug Fixes

astropy.convolution

- Fixed bug in `convolve_fft` where masked input was copied with `numpy.asarray` instead of `numpy.asanyarray`. `numpy.asarray` removes the mask subclass causing `numpy.ma.ismasked(input)` to fail, causing `convolve_fft` to ignore all masked input. [#8137]
- Remove function side-effects of input data from `convolve_fft`. It was possible for input data to remain modified if particular exceptions were raised. [#8152]

astropy.coordinates

- `EarthLocation.of_address` now uses the OpenStreetMap geocoding API by default to retrieve coordinates, with the Google API (which now

requires an API key) as an option. [#7918]
- Fixed a bug that caused frame objects with NaN distances to have NaN sky positions, even if valid sky coordinates were specified. [#7988]
- Fixed `represent_as()` to not round-trip through cartesian if the same representation class as the instance is passed in. [#7988]

astropy.io.ascii

- Fixed a problem when `guess=True` that `fast_reader` options could be dropped after the first fast reader class was tried. [#5578]
- Units in CDS-formatted tables are now parsed correctly by the units module. [#7348]

astropy.io.misc

- Fixed bug when writing a table with masked columns to HDF5. Previously the mask was being silently dropped. If the `serialize_meta` option is enabled the data mask will now be written as an additional column and the masked columns will round-trip correctly. [#7481]
- Fixed a bug where writing to HDF5 failed for for tables with columns of unicode strings. Now those columns are first encoded to UTF-8 and written as byte strings. [#7024, #8017]
- Fixed a bug with serializing the bounding_box of models initialized with `Quantities` . [#8052]

astropy.io.fits

- Added support for `copy.copy` and `copy.deepcopy` for `HDUList`. [#7218]
- Override `HDUList.copy()` to return a shallow HDUList instance. [#7218]

astropy.modeling

- Fix behaviour of certain models with units, by making certain unit-related attributes readonly. [#7210]
- Fixed an issue with validating a `bounding_box` whose items are `Quantities`. [#8052]
- Fix `Moffat1D` and `Moffat2D` derivatives. [#8108]

astropy.nddata

- Fixed rounding behavior in `overlap_slices` for even-sized small arrays. [#7859]
- Added support for pickling `NDData` instances that have an uncertainty. [#7383]

astropy.stats

- Fix errors in `kuiper_false_positive_probability`. [#7975]

astropy.tests

- Fixing bug that prevented to run the doctests on only a single rst documentation file rather than all of them. [#8055]

astropy.time

- Fix a bug when setting a `TimeDelta` array item with plain float value(s). This was always interpreted as a JD (day) value regardless of the `TimeDelta` format. [#7990]

astropy.units

- To simplify fast creation of `Quantity` instances from arrays, one can now write `array << unit` (equivalent to `Quantity(array, unit, copy=False)`). If `array` is already a `Quantity`, this will convert the quantity to the requested units; in-place conversion can be done with `quantity <<= unit`. [#7734]

astropy.utils

- Fixed a bug due to which `report_diff_values()` was reporting incorrect number of differences when comparing two `numpy.ndarray`. [#7470]
- The download progress bar is now only displayed in terminals, to avoid polluting piped output. [#7577]
- Ignore URL mirror caching when there is no internet. [#8163]

astropy.visualization

- Right ascension coordinates are now shown in hours by default, and the `set_format_unit` method on `CoordinateHelper` now works correctly with angle coordinates. [#7215]

Other Changes and Additions

- The documentation build now uses the Sphinx configuration from sphinx-astropy rather than from astropy-helpers. [#7139]
- Versions of Numpy <1.13 are no longer supported. [#7058]
- Running tests now suppresses the output of the installation stage by default, to allow easier viewing of the test results. To re-enable the output as before, use `python setup.py test --verbose-install`. [#7512]

- The ERFA functions are now wrapped in ufuncs instead of custom C code, leading to some speed improvements, and setting the stage for allowing overrides with `__array_ufunc__`. [#7502]
- Updated the bundled CFITSIO library to 3.450. See `cextern/cfitsio/docs/changes.txt` for additional information. [#8014]
- The `representation` keywords in coordinate frames are now deprecated in favor of the `representation_type` keywords (which are less ambiguously named). [#8119]

3.0.5 (2018-10-14)

Bug Fixes

astropy.coordinates

- Fixed bug in which consecutive `StaticMatrixTransform`'s in a frame transform path would be combined in the incorrect order. [#7707]

astropy.tests

- Fixing bug that doctests were not picked up from the narrative documentation when tests were run for all modules. [#7767]

3.0.4 (2018-08-02)

API Changes

astropy.table

- The private `_parent` attribute in the `info` attribute of table columns was changed from a direct reference to the parent column to a weak reference. This was in response to a memory leak caused by having a circular reference cycle. This change means that expressions like `col[3:5].info` will now fail because at the point of the `info` property being evaluated the `col[3:5]` weak reference is dead. Instead force a reference with `c = col[3:5]` followed by `c.info.indices`. [#6277, #7448]

Bug Fixes

astropy.nddata

- Fixed an bug when creating the `WCS` of a cutout (see `nddata.Cutout2D`) when input image's `WCS` contains `SIP` distortion corrections by adjusting the `crpix` of the `astropy.wcs.Sip` (in addition to adjusting the `crpix` of the `astropy.wcs.WCS` object). This bug had the potential to produce large errors in `WCS` coordinate transformations depending on the position of the cutout relative to the input image's `crpix`. [#7556, #7550]

astropy.table

- Fix memory leak where updating a table column or deleting a table object was not releasing the memory due to a reference cycle in the column `info` attributes. [#6277, #7448]

astropy.wcs

- Fixed an bug when creating the `WCS` slice (see `WCS.slice()`) when `WCS` contains `SIP` distortion corrections by adjusting the `WCS.sip.crpix` in addition to adjusting `WCS.wcs.crpix`. This bug had the potential to produce large errors in `WCS` coordinate transformations depending on the position of the slice relative to `WCS.wcs.crpix`. [#7556, #7550]

Other Changes and Additions

- Updated bundled wcslib to v 5.19.1 [#7688]

3.0.3 (2018-06-01)

Bug Fixes

astropy.io.ascii

- Fix stripping correct (header) comment line from `meta['comments']` in the `CommentedHeader` reader for all `header_start` settings. [#7508]

astropy.io.fits

- Raise error when attempting to open gzipped FITS file in 'append' mode. [#7473]
- Fix a bug when writing to FITS a table that has a column description with embedded blank lines. [#7482]

astropy.tests

- Enabling running tests for multiple packages when specified comma separated. [#7463]

3.0.2 (2018-04-23)

Bug Fixes

astropy.coordinates

- Computing a 3D separation between two `SkyCoord` objects (with the `separation_3d` method) now works with or without velocity data attached to the objects. [#7387]

astropy.io.votable

- Fix validate with xmllint=True. [#7255, #7283]

astropy.modeling

- `FittingWithOutlierRemoval` now handles model sets, as long as the underlying fitter supports masked values. [#7199]
- Remove assumption that `model_set_axis == 0` for 2D models in `LinearLSQFitter`. [#7317, #7199]
- Fix the shape of the outputs when a model set is evaluated with `model_set_axis=False` . [#7317]

astropy.stats

- Accept a tuple for the `axis` parameter in `sigma_clip`, like the underlying `numpy` functions and some other functions in `stats`. [#7199]

astropy.tests

- The function `quantity_allclose` was moved to the `units` package with the new, shorter name `allclose`. This eliminates a runtime dependency on `pytest` which was causing issues for some affiliated packages. The old import will continue to work but may be deprecated in the future. [#7252]

astropy.units

- Added a units-aware `allclose` function (this was previously available in the `tests` module as `quantity_allclose`). To complement `allclose`, a new `isclose` function is also added and backported. [#7252]

3.0.1 (2018-03-12)

Bug Fixes

astropy.io.ascii

- Fix a unicode decode error when reading a table with non-ASCII characters. The fast C reader cannot handle unicode so the code now uses the pure-Python reader in this case. [#7103]

astropy.io.fits

- Updated the bundled CFITSIO library to 3.430. This is to remedy a critical security vulnerability that was identified by NASA. See `cextern/cfitsio/docs/changes.txt` for additional information. [#7274]

astropy.io.misc

- Make sure that a sufficiently recent version of ASDF is installed when running test suite against ASDF tags and schemas. [#7205]

astropy.io.registry

- Fix reading files with serialized metadata when using a Table subclass. [#7213]

astropy.io.votable

- Fix lookup fields by ID. [#7208]

astropy.modeling

- Fix model set evaluation over common input when model_set_axis > 0. [#7222]
- Fixed the evaluation of compound models with units. This required adding the ability to have `input_units_strict` and `input_units_allow_dimensionless` be dictionaries with input names as keys. [#6952]

astropy.units

- `quantity_helper` no longer requires `scipy>=0.18`. [#7219]

3.0 (2018-02-12)

New Features

astropy.constants

- New context manager `set_enabled_constants` to temporarily use an older version. [#7008]

astropy.coordinates

- The `Distance` object now accepts `parallax` as a keyword in the initializer, and supports retrieving a parallax (as an `Angle`) via the `.parallax` attributes. [#6855]
- The coordinate frame classes (subclasses of `BaseCoordinateFrame`) now always have `.velocity`, `.proper_motion`, and `.radial_velocity` properties that provide shorthands to the full-space Cartesian velocity as a `CartesianDifferential`, the 2D proper motion as a `Quantity`, and the radial or line-of-sight velocity as a `Quantity`. [#6869]
- `SkyCoord` objects now support storing and tranforming differentials - i.e., both radial velocities and proper motions. [#6944]
- All frame classes now automatically get sensible representation mappings for velocity components. For example, `d_x`, `d_y`, `d_z` are all automatically mapped to frame component namse `v_x`, `v_y`, `v_z`. [#6856]
- `SkyCoord` objects now support updating the position of a source given its space motion and a new time or time difference. [#6872]
- The frame classes now accept a representation class or differential class, or string names for either, through the keyword arguments `representation_type` and `differential_type` instead of `representation` and `differential_cls`. [#6873]
- The frame classes (and `SkyCoord`) now give more useful error messages when incorrect attribute names are given. Instead of using the representation attribute names, they use the frame attribute names. [#7106]
- `EarthLocation` now has a method to compute the gravitational redshift due due to solar system bodies. [#6861, #6935]
- `EarthLocation` now has a `get_gcrs` convenience method to get the location in GCRS coordinates. [#6861, #6935]

astropy.io.fits

- Expanded the FITS `Column` interface to accept attributes pertaining to the FITS World Coordinate System, which includes spatial(celestial) and time coordinates. [#6359]
- Added `ver` attribute to set the `EXTVER` header keyword to `ImageHDU` and `TableHDU`. [#6454]
- The performance for reading FITS tables has been significantly improved, in

particular for cases where the tables contain one or more string columns and when done through `Table.read`. [#6821]
- The performance for writing tables from `Table.write` has now been significantly improved for tables containing one or more string columns. [#6920]
- The `Table.read` now supports a `memmap=` keyword argument to control whether or not to use memory mapping when reading the table. [#6821]
- When reading FITS tables with `fits.open`, a new keyword argument `character_as_bytes` can be passed - when set to **`True`**, character columns are returned as Numpy byte arrays (Numpy type S) while when set to **`False`**, the same columns are decoded to Unicode strings (Numpy type U) which uses more memory. [#6821]
- The `table_to_hdu` function and the `BinTableHDU.from_columns` and `FITS_rec.from_columns` methods now include a `character_as_bytes` keyword argument - if set to **`True`**, then when string columns are accessed, byte columns will be returned, which can provide significantly improved performance. [#6920]
- Added support for writing and reading back a table which has “mixin columns” such as `SkyCoord` or `EarthLocation` with no loss of information. [#6912]
- Enable tab-completion for `FITS_rec` column names and `Header` keywords with IPython 5 and later. [#7071]

astropy.io.misc

- When writing to HDF5 files, the serialized metadata are now saved in a new dataset, instead of the HDF5 dataset attributes. This allows for metadata of any dimensions. [#6304]
- Added support in HDF5 for writing and reading back a table which has “mixin columns” such as `SkyCoord` or `EarthLocation` with no loss of information. [#7007]
- Add implementations of astropy-specific ASDF tag types. [#6790]
- Add ASDF tag and schema for ICRSCoord. [#6904]

astropy.modeling

- Add unit support for tabular models. [#6529]
- A `deepcopy()` method was added to models. [#6515]
- Added units support to `AffineTransformation`. [#6853]
- Added `is_separable` function to modeling to test the separability of a model. [#6746]
- Added `Model.separable` property. It returns a boolean value or `None` if not set. [#6746]

- Support masked array values in `LinearLSQFitter` (instead of silently ignorning the mask). [#6927]

astropy.stats

- Added false alarm probability computation to `astropy.stats.LombScargle` [#6488]
- Implemented Kuiper functions in `astropy.stats` [#3724, #6565]

astropy.table

- Added support for reading and writing `astropy.time.Time` Table columns to and from FITS tables, to the extent supported by the FITS standard. [#6176]
- Improved exception handling and error messages when column `format` attribute is incorrect for the column type. [#6385]
- Allow to pass `htmldict` option to the jsviewer writer. [#6551]
- Added new table operation `astropy.table.setdiff` that returns the set difference of table rows for two tables. [#6443]
- Added support for reading time columns in FITS compliant binary tables as `astropy.time.Time` Table columns. [#6442]
- Allowed to remove table rows through the `__delitem__` method. [#5839]
- Added a new `showtable` command-line script to view binary or ASCII table files. [#6859]
- Added new table property `astropy.table.Table.loc_indices` that returns the location of rows by indexes. [#6831]
- Allow updating of table by indices through the property `astropy.table.Table.loc`. [#6831]
- Enable tab-completion for column names with IPython 5 and later. [#7071]
- Allow getting and setting a table Row using multiple column names. [#7107]

astropy.tests

- Split pytest plugins into separate modules. Move remotedata, openfiles, doctestplus plugins to standalone repositories. [#6384, #6606]
- When testing, astropy (or the package being tested) is now installed to a temporary directory instead of copying the build. This allows entry points to work correctly. [#6890]
- The tests_require setting in setup.py now works properly when running 'python setup.py test'. [#6892]

astropy.units

- Deprecated conversion of quantities to truth values. Currently, the expression

`bool(0 * u.dimensionless_unscaled)` evaluates to `True`. In the future, attempting to convert a `Quantity` to a `bool` will raise `ValueError`. [#6580, #6590]
- Modify the `brightness_temperature` equivalency to provide a surface brightness equivalency instead of the awkward assumed-per-beam equivalency that previously existed [#5173, #6663]
- Support was added for a number of `scipy.special` functions. [#6852]

astropy.utils

- The `astropy.utils.console.ProgressBar.map` class method now supports the `ipython_widget` option. You can now pass it both `multiprocess=True` and `ipython_widget=True` to get both multiprocess speedup and a progress bar widget in an IPython Notebook. [#6368]
- The `astropy.utils.compat.funcsigs` module has now been deprecated. Use the Python 'inspect' module directly instead. [#6598]
- The `astropy.utils.compat.futures` module has now been deprecated. Use the Python 'concurrent.futures' module directly instead. [#6598]
- `JsonCustomEncoder` is expanded to handle `Quantity` and `UnitBase`. [#5471]
- Added a `dcip_xy` method to IERS that interpolates along the dX_2000A and dY_2000A columns of the IERS table. Hence, the data for the CIP offsets is now available for use in coordinate frame conversion. [#5837]
- The functions `matmul`, `broadcast_arrays`, `broadcast_to` of the `astropy.utils.compat.numpy` module have been deprecated. Use the NumPy functions directly. [#6691]
- The `astropy.utils.console.ProgressBar.map` class method now returns results in sequential order. Previously, if you set `multiprocess=True`, then the results could arrive in any arbitrary order, which could be a nasty shock. Although the function will still be evaluated on the items in arbitrary order, the return values will arrive in the same order in which the input items were provided. The method is now a thin wrapper around `astropy.utils.console.ProgressBar.map_unordered`, which preserves the old behavior. [#6439]

astropy.visualization

- Enable Matplotlib's subtraction shorthand syntax for composing and inverting trasformations for the `WCSWorld2PixelTransform` and `WCSPixel2WorldTransform` classes by setting `has_inverse` to

`True`. In order to implement a unit test, also implement the equality comparison operator for both classes. [#6531]
- Added automatic hiding of axes labels when no tick labels are drawn on that axis. This parameter can be configured with `WCSAxes.coords[*].set_axislabel_visibility_rule` so that labels are automatically hidden when no ticks are drawn or always shown. [#6774]

astropy.wcs

- Added a new function `celestial_frame_to_wcs` to convert from coordinate frames to WCS (the opposite of what `wcs_to_celestial_frame` currently does. [#6481]
- `wcslib` was updated to v 5.18. [#7066]

API Changes

astropy.convolution

- `Gaussian2DKernel` now accepts `x_stddev` in place of `stddev` with an option for `y_stddev`, if different. It also accepts `theta` like `Gaussian2D` model. [#3605, #6748]

astropy.coordinates

- Deprecated `recommended_units` for representations. These were used to ensure that any angle was presented in degrees in sky coordinates and frames. This is more logically done in the frame itself. [#6858]
- As noted above, the frame class attributes `representation` and `differential_cls` are being replaced by `representation_type` and `differential_type`. In the next version, using `representation` will raise a deprecation warning. [#6873]
- Coordinate frame classes now can’t be added to the frame transform graph if they have frame attribute names that conflict with any component names. This is so `SkyCoord` can uniquely identify and distinguish frame attributes from frame components. [#6871]
- Slicing and reshaping of `SkyCoord` and coordinate frames no longer passes the new object through `__init__`, but directly sets atttributes on a new instance. This speeds up those methods by an order of magnitude, but means that any customization done in `__init__` is by-passed. [#6941]

astropy.io.ascii

- Allow ECSV files to be auto-identified by `Table.read` or `Table.write` based on the `.ecsv` file name suffix. In this case it is not required to provide the `format` keyword. [#6552]

astropy.io.fits

- Automatically detect and handle compression in FITS files that are opened by passing a file handle to `fits.open` [#6373]
- Remove the `nonstandard` checksum option. [#6571]

astropy.io.misc

- When writing to HDF5 files, the serialized metadata are now saved in a new dataset instead of the HDF5 dataset attributes. This allows for metadata of any dimensions. [#6304]
- Deprecated the `usecPickle` kwarg of `fnunpickle` and `fnpickle` as it was needed only for Python2 usage. [#6655]

astropy.io.votable

- Add handling of `tree.Group` elements to `tree.Resource`. Unified I/O or conversion to astropy tables is not affected. [#6262]

astropy.modeling

- Removed deprecated `GaussianAbsorption1D` model. Use `Const1D - Gaussian1D` instead. [#6542]
- Removed the registry from modeling. [#6706]

astropy.table

- When setting the column `format` attribute the value is now immediately validated. Previously one could set to any value and it was only checked when actually formatting the column. [#6385]
- Deprecated the `python3_only` kwarg of the `convert_bytestring_to_unicode` and `convert_unicode_to_bytestring` methods it was needed only for Python2 usage. [#6655]
- When reading in FITS tables with `Table.read`, string columns are now represented using Numpy byte (dtype `S`) arrays rather than Numpy unicode arrays (dtype `U`). The `Column` class then ensures the bytes are automatically converted to string as needed. [#6821]
- When getting a table row using multiple column names, if one of the names is not a valid column name then a `KeyError` exception is now raised (previously `ValueError`). When setting a table row, if the right hand side

is not a sequence with the correct length then a `ValueError` is now raised (previously in certain cases a `TypeError` was raised). [#7107]

astropy.utils

- `download_files_in_parallel` now always uses `cache=True` to make the function work on Windows. [#6671]

astropy.visualization

- The Astropy matplotlib plot style has been deprecated. It will continue to work in future but is no longer documented. [#6991]

Bug Fixes

astropy.coordinates

- Frame objects now use the default differential even if the representation is explicitly provided as long as the representation provided is the same type as the default representation. [#6944]
- Coordinate frame classes now raise an error when they are added to the frame transform graph if they have frame attribute names that conflict with any component names. [#6871]

astropy.io.ascii

- Added support for reading very large tables in chunks to reduce memory usage. [#6458]
- Strip leading/trailing white-space from latex lines to avoid issues when matching `\begin{tabular}` statements. This is done by introducing a new `LatexInputter` class to override the `BaseInputter`. [#6311]

astropy.io.fits

- Properly handle opening of FITS files from `http.client.HTTPResponse` (i.e. it now works correctly when passing the results of `urllib.request.urlopen` to `fits.open`). [#6378]
- Fix the `fitscheck` script for updating invalid checksums, or removing checksums. [#6571]
- Fixed potential problems with the compression module [#6732]
- Always use the ‘D’ format for floating point values in ascii tables. [#6938]

astropy.table

- Fix getting a table row when using multiple column names (for example `t[3]['a', 'b', 'c']`). Also fix a problem when setting an entire row: if setting one of the right-hand side values failed this could result in a partial update of the referenced parent table before the exception is raised. [#7107]

astropy.time

- Initialization of `Time` instances with bytes or arrays with dtype `S` will now automatically attempt to decode as ASCII. This ensures `Column` instances with ASCII strings stored with dtype `S` can be used. [#6823, #6903]

astropy.units

- Fixed a bug that caused PLY files to not be generated correctly in Python 3. [#7174]

astropy.utils

- The `deprecated` decorator applied to a class will now modify the class itself, rather than to create a class that just looks and behave like the original. This is needed so that the Python 3 `super` without arguments works for decorated classes. [#6615]
- Fixed `HomogeneousList` when setting one item or a slice. [#6773]
- Also check the type when creating a new instance of `HomogeneousList`. [#6773]
- Make `HomogeneousList` work with iterators and generators when creating the instance, extending it, or using when setting a slice. [#6773]

Other Changes and Additions

- Versions of Python <3.5 are no longer supported. [#6556]
- Versions of Pytest <3.1 are no longer supported. [#6419]
- Versions of Numpy <1.10 are no longer supported. [#6593]
- The bundled CFITSIO was updated to version 3.41 [#6477]
- `analytic_functions` sub-package is removed. Use `astropy.modeling.blackbody`. [#6541]
- `astropy.vo` sub-package is removed. Use `astropy.samp` for SAMP and `astroquery` for VO cone search. [#6540]
- The guide to setting up Emacs for code development was simplified, and updated to recommend `flycheck` and `flake8` for syntax checks. [#6692]
- The bundled version of PLY was updated to 3.10. [#7174]

2.0.16 (2019-10-27)

Bug Fixes

astropy.time

- Fixed a troubling bug in which `Time` could loose precision, with deviations of 300 ns. [#9328]

Other Changes and Additions

- Updated IERS A URLs due to USNO prolonged maintenance. [#9443]

2.0.15 (2019-10-06)

Bug Fixes

astropy.coordinates

- Fixed a bug where the string representation of a `BaseCoordinateFrame` object could become garbled under specific circumstances when the frame defines custom component names via `RepresentationMapping`. [#8869]

astropy.io.fits

- Fix uint conversion in `FITS_rec` when slicing a table. [#8982]
- Fix reading of unsigned 8-bit integer with compressed fits. [#9219]

astropy.nddata

- Fixed a bug in `overlap_slices` where the `"strict"` mode was too strict for a small array along the upper edge of the large array. [#8901]
- Fixed a bug in `overlap_slices` where a `NoOverlapError` would be incorrectly raised for a 0-shaped small array at the origin. [#8901]

astropy.samp

- Fixed a bug that caused an incorrectly constructed warning message to raise an error. [#8966]

astropy.table

- Fix `FixedWidthNoHeader` to pay attention to `data_start` keyword when finding first data line to split columns [#8485, #8511]
- Fix bug when initializing `Table` with `rows` as a generator. [#9315]
- Fix `join` when there are multiple mixin (Quantity) columns as keys. [#9313]

astropy.units

- `Quantity` now preserves the `dtype` for anything that is floating point, including `float16`. [#8872]
- `Unit()` now accepts units with fractional exponents such as `m(3/2)` in the default/`fits` and `vounit` formats that would previously have been rejected for containing multiple solidi (`/`). [#9000]
- Fixed the LaTeX representation of units containing a superscript. [#9218]

astropy.visualization

- Fixed compatibility issues with latest versions of Matplotlib. [#8961]

Other Changes and Additions

- Updated required version of Cython to v0.29.13 to make sure that generated C files are compatible with the upcoming Python 3.8 release as well as earlier supported versions of Python. [#9198]

2.0.14 (2019-06-14)

Bug Fixes

astropy.io.fits

- Fix `Header.update` which was dropping the comments when passed a `Header` object. [#8840]

astropy.modeling

- `Moffat1D.fwhm` and `Moffat2D.fwhm` will return a positive value when `gamma` is negative. [#8801, #8815]

astropy.units

- Fixed a bug that prevented `EarthLocation` from being initialized with numpy >=1.17. [#8849]

astropy.visualization

- Fixed `quantity_support` to work around the fact that matplotlib does not detect subclasses in its `units` framework. With this, `Angle` and other subclasses work correctly. [#8818]
- Fixed `quantity_support` to work properly if multiple context managers are nested. [#8844]

2.0.13 (2019-06-08)

Bug Fixes

astropy.io.fits

- Fixed bug in `ColDefs._init_from_array()` that caused unsigned datatypes with the opposite endianess as the host architecture to fail the TestColumnFunctions.test_coldefs_init_from_array unit test. [#8460]

astropy.io.misc

- Explicitly set PyYAML default flow style to None to ensure consistent astropy YAML output for PyYAML version 5.1 and later. [#8500]

astropy.io.votable

- Block floating-point columns from using repr format when converted to Table [#8358]

astropy.stats

- Fixed issue in `bayesian_blocks` when called with the `ncp_prior` keyword. [#8339]

astropy.units

- Fix `take` when one gets only a single element from a `Quantity`, ensuring it returns a `Quantity` rather than a scalar. [#8617]

2.0.12 (2019-02-23)

New Features

astropy.utils

- The `deprecated_renamed_argument` decorator now capable deprecating an argument without renaming it. It also got a new `alternative` keyword argument to suggest alternative functionality instead of the removed one. [#8324]

Bug Fixes

astropy.io.fits

- Fixed bug in `ColDefs._init_from_array()` that caused non-scalar unsigned entries to not have the correct bzero value set. [#8353]

astropy.modeling

- Fixed compatibility of `JointFitter` with the latest version of Numpy. [#7984]

astropy.table

- Fix `.quantity` property of `Column` class for function-units (e.g., `dex`). Previously setting this was possible, but getting raised an error. [#8425]
- Fixes a bug where initializing a new `Table` from the final row of an existing `Table` failed. This happened when that row was generated using the item index `[-1]`. [#8422]

astropy.wcs

- Fix bug that caused `WCS.has_celestial`, `wcs_to_celestial_frame`, and other functionality depending on it to fail in the presence of correlated celestial and other axes. [#8420]

Other Changes and Additions

- Fixed `make clean` for the documentation on Windows to ensure it properly removes the `api` and `generated` directories. [#8346]
- Updating bundled `pytest-openfiles` to v0.3.2. [#8434]
- Making `ErfaWarning` and `ErfaError` available via `astropy.utils.exceptions`. [#8441]

2.0.11 (2018-12-31)

Bug Fixes

astropy.io.ascii

- Fix fast reader C tokenizer to handle double quotes in quoted field. [#8283]

astropy.io.fits

- Fix a bug in `io.fits` with writing Fortran-ordered arrays to file objects. [#8282]

astropy.units

- Add support for `np.matmul` as a `ufunc` (new in numpy 1.16). [#8264, #8305]

astropy.utils

- Fix failures caused by IERS_A_URL being unavailable by introducing IERS_A_URL_MIRROR. [#8308]

2.0.10 (2018-12-04)

Bug Fixes

astropy.convolution

- Fix Moffat2DKernel's FWHM computation, which has an influence on the default size of the kernel when no size is given. [#8105]

astropy.coordinates

- Disable `of_address` usage due to Google API now requiring API key. [#7993]

astropy.io.fits

- `fits.append` now correctly handles file objects with valid modes other than `ostream`. [#7856]

astropy.table

- Fix `Table.show_in_notebook` failure when mixin columns are present. [#8069]

astropy.tests

- Explicitly disallow incompatible versions of `pytest` when using the test runner. [#8188]

astropy.units

- Fixed the spelling of the 'luminous emittance/illuminance' physical property. [#7942]

astropy.visualization

- Fixed a bug that caused origin to be incorrect if not specified. [#7927]
- Fixed a bug that caused an error when plotting grids multiple times with grid_type='contours'. [#7927]

- Put an upper limit on the number of bins in `hist` and `histogram` and factor out calculation of bin edges into public function `calculate_bin_edges`. [#7991]

Other Changes and Additions

- Fixing `astropy.__citation__` to provide the full bibtex entry of the 2018 paper. [#8110]
- Pytest 4.0 is not supported by the 2.0.x LTS releases. [#8173]
- Updating bundled `pytest-remotedata` to v0.3.1. [#8174]
- Updating bundled `pytest-doctestplus` to v0.2.0. [#8175]
- Updating bundled `pytest-openfiles` to v0.3.0. [#8176]
- Adding `warning_type` keyword argument to the “deprecated” decorators to allow issuing custom warning types instead of the default `AstropyDeprecationWarning`. [#8178]

2.0.9 (2018-10-14)

Bug Fixes

astropy.io.ascii

- Fix reading of big files with the fast reader. [#7885]

astropy.io.fits

- `HDUList.__contains__()` now works with `HDU` arguments. That is, `hdulist[0] in hdulist` now works as expected. [#7282]
- `HDUList` s can now be written to streams in Python 3 [#7850]

astropy.nddata

- Fixed the bug in CCData.read when the HDU is not specified and the first one is empty so the function searches for the first HDU with data which may not have an image extension. [#7739]

astropy.stats

- Fixed bugs in biweight statistics functions where a constant data array (or if using the axis keyword, constant along an axis) would return NaN. [#7737]

astropy.table

- Fixed a bug in `to_pandas()` where integer type masked columns were always getting converted to float. This could cause loss of precision. Now this only occurs if there are actually masked data values, in which case `pandas` does require the values to be float so that `NaN` can be used to mark the masked values. [#7741, #7747]

astropy.tests

- Change the name of the configuration variable controlling the location of the Astropy cache in the Pytest plugin from `cache_dir` to `astropy_cache_dir`. The command line flag also changed to `--astropy-cache-dir`. This prevents a conflict with the `cache_dir` variable provided by pytest itself. Also made similar change to `config_dir` option as a precaution. [#7721]

astropy.units

- `UnrecognizedUnit` instances can now be compared to any other object without raising **`TypeError`**. [#7606]

astropy.visualization

- Fix compatibility with Matplotlib 3.0. [#7839]
- Fix an issue that caused a crash when using WCSAxes with a custom Transform object and when using `grid_type='contours'` to plot a grid. [#7846]

astropy.wcs

- Instead of raising an error `astropy.wcs` now returns the input when the input has zero size. [#7746]
- Fix `malloc(0)` bug in `pipeline_all_pixel2world()` and `pipeline_pix2foc()`. They now raise an exception for input with zero coordinates, i.e. shape = (0, n). [#7806]
- Fixed an issue with scalar input when WCS.naxis is one. [#7858]

Other Changes and Additions

- Added a new `astropy.__citation__` attribute which gives a citation for Astropy in bibtex format. Made sure that both this and `astropy.__bibtex__` works outside the source environment, too. [#7718]

2.0.8 (2018-08-02)

Bug Fixes

astropy.convolution

- Correct data type conversion for non-float masked kernels. [#7542]
- Fix non-float or masked, zero sum kernels when `normalize_kernel=False`. Non-floats would yeild a type error and masked kernels were not being filled. [#7541]

astropy.coordinates

- Ensure that relative humidities can be given as Quantities, rather than take any quantity and just strip its unit. [#7668]

astropy.nddata

- Fixed `Cutout2D` output WCS NAXIS values to reflect the cutout image size. [#7552]

astropy.table

- Fixed a bug in `add_columns` method where `rename_duplicate=True` would cause an error if there were no duplicates. [#7540]

astropy.tests

- Fixed bug in `python setup.py test --coverage` on Windows machines. [#7673]

astropy.time

- Avoid rounding errors when converting `Quantity` to `TimeDelta`. [#7625]

astropy.visualization

- Fixed a bug that caused the position of the tick values in decimal mode to be incorrectly determined. [#7332]

astropy.wcs

- Fixed a bug that caused `wcs_to_celestial_frame`, `skycoord_to_pixel`, and `pixel_to_skycoord` to raise an error if the axes of the celestial WCS were swapped. [#7691]

2.0.7 (2018-06-01)

Bug Fixes

astropy.modeling

- Fixed `Tabular` models to not change the shape of data. [#7411]

astropy.stats

- In `freedman_bin_width`, if the data has too small IQR, raise `ValueError`. [#7248, #7402]

astropy.table

- Fix a performance issue in `MaskedColumn` where initialization was extremely slow for large arrays with the default `mask=None`. [#7422]
- Fix printing table row indexed with unsigned integer. [#7469]
- Fix copy of mask when copying a Table, as this is no more done systematically by Numpy since version 1.14. Also fixed a problem when MaskedColumn was initialized with `mask=np.ma.nomask`. [#7486]

astropy.time

- Fixed a bug in Time that raised an error when initializing a subclass of Time with a Time object. [#7453]

astropy.utils

- Fixed a bug that improperly handled unicode case of URL mirror in Python 2. [#7493]

astropy.visualization

- Fixed a bug that prevented legends from being added to plots done with units. [#7510]

Other Changes and Additions

- Bundled `pytest-remotedata` plugin is upgraded to 0.3. [#7493]

2.0.6 (2018-04-23)

Bug Fixes

astropy.convolution

- convolve(boundary=None) requires the kernel to be smaller than the image. This was never actually checked, it now is and an exception is raised. [#7313]

astropy.units

- `u.quantity_input` no longer errors if the return annotation for a function is `None`. [#7336, #7380]

astropy.visualization

- Explicitly default to origin='lower' in WCSAxes. [#7331]
- Lists of units are now converted in the Matplotlib unit converter. This means that for Matplotlib versions later than 2.2, more plotting functions now work with units (e.g. errorbar). [#7037]

Other Changes and Additions

- Updated the bundled CFITSIO library to 3.44. This is to remedy another critical security vulnerability that was identified by NASA. See `cextern/cfitsio/docs/changes.txt` for additional information. [#7370]

2.0.5 (2018-03-12)

Bug Fixes

astropy.coordinates

- Add a workaround for a bug in the einsum function in Numpy 1.14.0. [#7187]
- Fix problems with printing `Angle` instances under numpy 1.14.1. [#7234]

astropy.io.fits

- Fixed the `fitsdiff` script for matching fits file with one in a directory path. [#7085]
- Make sure that lazily-loaded `HDUList` is automatically loaded when calling `hdulist.pop`. [#7186]

astropy.modeling

- Propagate weights to underlying fitter in `FittingWithOutlierRemoval` [#7249]

astropy.tests

- Support dotted package names as namespace packages when gathering test coverage. [#7170]

astropy.visualization

- Matplotlib axes have the `axisbelow` property to control the z-order of ticks, tick labels, and grid lines. WCSAxes will now respect this property. This is useful for drawing scale bars or inset boxes, which should have a z-order that places them above all ticks and gridlines. [#7098]

Other Changes and Additions

- Updated the bundled CFITSIO library to 3.430. This is to remedy a critical security vulnerability that was identified by NASA. See `cextern/cfitsio/docs/changes.txt` for additional information. [#7274, #7275]

2.0.4 (2018-02-06)

Bug Fixes

astropy.convolution

- Fixed IndexError when `preserve_nan=True` in `convolve_fft`. Added testing with `preserve_nan=True`. [#7000]

astropy.coordinates

- The `sites.json` file is now parsed explicitly with a UTF-8 encoding. This means that future revisions to the file with unicode observatory names can be done without breaking the site registry parser. [#7082]
- Working around a bug in Numpy 1.14.0 that broke some coordinate transformations. [#7105]
- Fixed a bug where negative angles could be rounded wrongly when converting to a string with seconds omitted. [#7148]

astropy.io.fits

- When datafile is missing, fits.tabledump uses input file name to build output file name. Fixed how it gets input file name from HDUList. [#6976]
- Fix in-place updates to scaled columns. [#6956]

astropy.io.registry

- Fixed bug in identifying inherited registrations from multiple ancestors [#7156]

astropy.modeling

- Fixed a bug in `LevMarLSQFitter` when fitting 2D models with constraints. [#6705]

astropy.utils

- `download_file` function will check for cache downloaded from mirror URL first before attempting actual download if primary URL is unavailable. [#6987]

astropy.visualization

- Fixed test failures for `astropy.visualization.wcsaxes` which were due to local matplotlibrc files being taken into account. [#7132]

Other Changes and Additions

- Fixed broken links in the documentation. [#6745]
- Substantial performance improvement (potentially >1000x for some cases) when converting non-scalar `coordinates.Angle` objects to strings. [#7004]

2.0.3 (2017-12-13)

Bug Fixes

astropy.coordinates

- Ecliptic frame classes now support attributes `v_x`, `v_y`, `v_z` when using with a Cartesian representation. [#6569]
- Added a nicer error message when accidentally calling `frame.representation` instead of `frame.data` in the context of methods that use `._apply()`. [#6561]
- Creating a new `SkyCoord` from a list of multiple `SkyCoord` objects now yield the correct type of frame, and works at all for non-equatorial frames. [#6612]
- Improved accuracy of velocity calculation in `EarthLocation.get_gcrs_posvel`. [#6699]
- Improved accuracy of radial velocity corrections in `` SkyCoord.radial_velocity_correction` ``. [#6861]
- The precision of ecliptic frames is now much better, after removing the nutation from the rotation and fixing the computation of the position of the Sun. [#6508]

astropy.extern

- Version 0.2.1 of `pytest-astropy` is included as an external package. [#6918]

astropy.io.fits

- Fix writing the result of `fitsdiff` to file with `--output-file`. [#6621]
- Fix a minor bug where `FITS_rec` instances can not be indexed with tuples and other sequences that end up with a scalar. [#6955, #6966]

astropy.io.misc

- Fix `ImportError` when `hdf5` is imported first in a fresh Python interpreter in Python 3. [#6604, #6610]

astropy.nddata

- Suppress errors during WCS creation in CCDData.read(). [#6500]
- Fixed a problem with `CCDData.read` when the extension wasn't given and the primary HDU contained no `data` but another HDU did. In that case the header were not correctly combined. [#6489]

astropy.stats

- Fixed an issue where the biweight statistics functions would sometimes cause runtime underflow/overflow errors for float32 input arrays. [#6905]

astropy.table

- Fixed a problem when printing a table when a column is deleted and garbage-collected, and the format function caching mechanism happens to re-use the same cache key. [#6714]
- Fixed a problem when comparing a unicode masked column (on left side) to a bytes masked column (on right side). [#6899]
- Fixed a problem in comparing masked columns in bytes and unicode when the unicode had masked entries. [#6899]

astropy.tests

- Fixed a bug that causes tests for rst files to not be run on certain platforms. [#6555, #6608]
- Fixed a bug that caused the doctestplus plugin to not work nicely with the hypothesis package. [#6605, #6609]
- Fixed a bug that meant that the data.astropy.org mirror could not be used when using –remote-data=astropy. [#6724]
- Support compatibility with new `pytest-astropy` plugins. [#6918]
- When testing, astropy (or the package being tested) is now installed to a

temporary directory instead of copying the build. This allows entry points to work correctly. [#6890]

astropy.time

- Initialization of Time instances now is consistent for all formats to ensure that `-0.5 <= jd2 < 0.5`. [#6653]

astropy.units

- Ensure that `Quantity` slices can be set with objects that have a `unit` attribute (such as `Column`). [#6123]

astropy.utils

- `download_files_in_parallel` now respects the given `timeout` value. [#6658]
- Fixed bugs in remote data handling and also in IERS unit test related to path URL, and URI normalization on Windows. [#6651]
- Fixed a bug that caused `get_pkg_data_fileobj` to not work correctly when used with non-local data from inside packages. [#6724]
- Make sure `get_pkg_data_fileobj` fails if the URL can not be read, and correctly falls back on the mirror if necessary. [#6767]
- Fix the `finddiff` option in `find_current_module` to properly deal with submodules. [#6767]
- Fixed `pyreadline` import in `utils.console.isatty` for older IPython versions on Windows. [#6800]

astropy.visualization

- Fixed the vertical orientation of the `fits2bitmap` output bitmap image to match that of the FITS image. [#6844, #6969]
- Added a workaround for a bug in matplotlib so that the `fits2bitmap` script generates the correct output file type. [#6969]

Other Changes and Additions

- No longer require LaTeX to build the documentation locally and use mathjax instead. [#6701]
- Ensured that all tests use the Astropy data mirror if needed. [#6767]

2.0.2 (2017-09-08)

Bug Fixes

astropy.coordinates

- Ensure transformations via ICRS also work for coordinates that use cartesian representations. [#6440]
- Fixed a bug that was preventing `SkyCoord` objects made from lists of other coordinate objects from being written out to ECSV files. [#6448]

astropy.io.fits

- Support the `GZIP_2` FITS image compression algorithm as claimed in docs. [#6486]

astropy.io.votable

- Fixed a bug that wrote out VO table as version 1.2 instead of 1.3. [#6521]

astropy.table

- Fix a bug when combining unicode columns via join or vstack. The character width of the output column was a factor of 4 larger than needed. [#6459]

astropy.tests

- Fixed running the test suite using –parallel. [#6415]
- Added error handling for attempting to run tests in parallel without having the `pytest-xdist` package installed. [#6416]
- Fixed issue running doctests with pytest>=3.2. [#6423, #6430]
- Fixed issue caused by antivirus software in response to malformed compressed files used for testing. [#6522]
- Updated top-level config file to properly ignore top-level directories. [#6449]

astropy.units

- Quantity._repr_latex_ now respects precision option from numpy printoptions. [#6412]

astropy.utils

- For the `deprecated_renamed_argument` decorator, refer to the deprecation's caller instead of `astropy.utils.decorators`, to makes it easier to find where the deprecation warnings comes from. [#6422]

2.0.1 (2017-07-30)

Bug Fixes

astropy.constants

- Fixed Earth radius to be the IAU2015 value for the equatorial radius. The polar value had erroneously been used in 2.0. [#6400]

astropy.coordinates

- Added old frame attribute classes back to top-level namespace of `astropy.coordinates`. [#6357]

astropy.io.fits

- Scaling an image always uses user-supplied values when given. Added defaults for scaling when bscale/bzero are not present (float images). Fixed a small bug in when to reset `_orig_bscale`. [#5955]

astropy.modeling

- Fixed a bug in initializing compound models with units. [#6398]

astropy.nddata

- Updating CCDData.read() to be more flexible with inputs, don't try to delete keywords that are missing from the header. [#6388]

astropy.tests

- Fixed the test command that is run from `setuptools` to allow it to gracefully handle keyboard interrupts and pass them on to the `pytest` subprocess. This prompts `pytest` to teardown and display useful traceback and test information [#6369]

astropy.visualization

- Ticks and tick labels are now drawn in front of, rather than behind, gridlines in WCS axes. This improves legibility in situations where tick labels may be on the interior of the axes frame, such as the right ascension axis of an all-sky Aitoff or Mollweide projection. [#6361]

astropy.wcs

- Fix the missing wcskey part in _read_sip_kw, this will cause error when reading sip wcs while there is no default CRPIX1 CRPIX2 keywords and only CRPIX1n CRPIX2n in header. [#6372]

2.0 (2017-07-07)

New Features

astropy.constants

- Constants are now organized into version modules, with physical CODATA constants in the `codata2010` and `codata2014` sub-modules, and astronomical constants defined by the IAU in the `iau2012` and `iau2015` sub-modules. The default constants in `astropy.constants` in Astropy 2.0 have been updated from `iau2012` to `iau2015` and from `codata2010` to `codata2014`. The constants for 1.3 can be accessed in the `astropyconst13` sub-module and the constants for 2.0 (the default in `astropy.constants`) can also be accessed in the `astropyconst20` sub-module [#6083]
- The GM mass parameters recommended by IAU 2015 Resolution B 3 have been added as `GM_sun`, `GM_jup`, and `GM_earth`, for the Sun, Jupiter and the Earth. [#6083]

astropy.convolution

- Major change in convolution behavior and keyword arguments. Additional details are in the API section. [#5782]
- Convolution with un-normalized and un-normalizable kernels is now possible. [#5782]
- Add a new argument, `normalization_rtol`, to `convolve_fft`, allowing the user to specify the relative error tolerance in the normalization of the convolution kernel. [#5649, #5177]
- Models can now be convoluted using `convolve` or `convolve_fft`, which generates a regular compound model. [#6015]

astropy.coordinates

- Frame attributes set on `SkyCoord` are now always validated, and any ndarray-like operation (like slicing) will also be done on those. [#5751]
- Caching of all possible frame attributes was implemented. This greatly speeds up many `SkyCoord` operations. [#5703, #5751]
- A class hierarchy was added to allow the representation layer to store differentials (i.e., finite derivatives) of coordinates. This is intended to enable support for velocities in coordinate frames. [#5871]
- `replicate_without_data` and `replicate` methods were added to coordinate frames that allow copying an existing frame object with various reference or copy behaviors and possibly overriding frame attributes. [#6182]
- The representation class instances can now contain differential objects. This is primarily useful for internal operations that will provide support for

transforming velocity components in coordinate frames. [#6169]
- `EarthLocation.to_geodetic()` (and `EarthLocation.geodetic`) now return namedtuples instead of regular tuples. [#6237]
- `EarthLocation` now has `lat` and `lon` properties (equivalent to, but preferred over, the previous `latitude` and `longitude`). [#6237]
- Added a `radial_velocity_correction` method to `SkyCoord` to do compute barycentric and heliocentric velocity corrections. [#5752]
- Added a new `AffineTransform` class for coordinate frame transformations. This class supports matrix operations with vector offsets in position or any differential quantities (so far, only velocity is supported). The matrix transform classes now subclass from the base affine transform. [#6218]
- Frame objects now have experimental support for velocity components. Most frames default to accepting proper motion components and radial velocity, and the velocities transform correctly for any transformation that uses one of the `AffineTransform`-type transformations. For other transformations a finite-difference velocity transformation is available, although it is not as numerically stable as those that use `AffineTransform`-type transformations. [#6219, #6226]

astropy.io.ascii

- Allow to specify encoding in `ascii.read`, only for Python 3 and with the pure-Python readers. [#5448]
- Writing latex tables with only a `tabular` environment is now possible by setting `latexdict['tabletyle']` to `None`. [#6205]
- Allow ECSV format to support reading and writing mixin columns like `Time`, `SkyCoord`, `Latitude`, and `EarthLocation`. [#6181]

astropy.io.fits

- Checking available disk space before writing out file. [#5550, #4065]
- Change behavior to warn about units that are not FITS-compliant when writing a FITS file but not when reading. [#5675]
- Added absolute tolerance parameter when comparing FITS files. [#4729]
- New convenience function `printdiff` to print out diff reports. [#5759]
- Allow to instantiate a `BinTableHDU` directly from a `Table` object. [#6139]

astropy.io.misc

- YAML representer now also accepts numpy types. [#6077]

astropy.io.registry

- New functions to unregister readers, writers, and identifiers. [#6217]

astropy.modeling

- Added `SmoothlyBrokenPowerLaw1D` model. [#5656]
- Add `n_submodels` shared method to single and compound models, which allows users to get the number of components of a given single (compound) model. [#5747]
- Added a `name` setter for instances of `_CompoundModel`. [#5741]
- Added FWHM properties to Gaussian and Moffat models. [#6027]
- Added support for evaluating models and setting the results for inputs outside the bounding_box to a user specified `fill_value`. This is controlled by a new optional boolean keyword `with_bounding_box`. [#6081]
- Added infrastructure support for units on parameters and during model evaluation and fitting, added support for units on all functional, power-law, polynomial, and rotation models where this is appropriate. A new BlackBody1D model has been added. [#4855, #6183, #6204, #6235]

astropy.nddata

- Added an image class, `CCDData`. [#6173]

astropy.stats

- Added `biweight_midcovariance` function. [#5777]
- Added `biweight_scale` and `biweight_midcorrelation` functions. [#5991]
- `median_absolute_deviation` and `mad_std` have `ignore_nan` option that will use `np.ma.median` with nans masked out or `np.nanmedian` instead of `np.median` when computing the median. [#5232]
- Implemented statistical estimators for Ripley's K Function. [#5712]
- Added `SigmaClip` class. [#6206]
- Added `std_ddof` keyword option to `sigma_clipped_stats`. [#6066, #6207]

astropy.table

- Issue a warning when assigning a string value to a column and the string gets truncated. This can occur because numpy string arrays are fixed-width and silently drop characters which do not fit within the fixed width. [#5624, #5819]
- Added functionality to allow `astropy.units.Quantity` to be written as a

normal column to FITS files. [#5910]
- Add support for Quantity columns (within a `QTable`) in table `join()`, `hstack()` and `vstack()` operations. [#5841]
- Allow unicode strings to be stored in a Table bytestring column in Python 3 using UTF-8 encoding. Allow comparison and assignment of Python 3 `str` object in a bytestring column (numpy `'S'` dtype). If comparison with `str` instead of `bytes` is a problem (and `bytes` is really more logical), please open an issue on GitHub. [#5700]
- Added functionality to allow `astropy.units.Quantity` to be read from and written to a VOtable file. [#6132]
- Added support for reading and writing a table with mixin columns like `Time`, `SkyCoord`, `Latitude`, and `EarthLocation` via the ASCII ECSV format. [#6181]
- Bug fix for `MaskedColumn` insert method, where `fill_value` attribute was not being passed along to the copy of the `MaskedColumn` that was returned. [#7585]

astropy.tests

- `enable_deprecations_as_exceptions` function now accepts additional user-defined module imports and warning messages to ignore. [#6223, #6334]

astropy.units

- The `astropy.units.quantity_input` decorator will now convert the output to the unit specified as a return annotation under Python 3. [#5606]
- Passing a logarithmic unit to the `Quantity` constructor now returns the appropriate logarithmic quantity class if `subok=True`. For instance, `Quantity(1, u.dex(u.m), subok=True)` yields `<Dex 1.0 dex(m)>`. [#5928]
- The `quantity_input` decorator now accepts a string physical type in addition to of a unit object to specify the expected input `Quantity`'s physical type. For example, `@u.quantity_input(x='angle')` is now functionally the same as `@u.quantity_input(x=u.degree)`. [#3847]
- The `quantity_input` decorator now also supports unit checking for optional keyword arguments and accepts iterables of units or physical types for specifying multiple valid equivalent inputs. For example, `@u.quantity_input(x=['angle', 'angular speed'])` or `@u.quantity_input(x=[u.radian, u.radian/u.yr])` would both allow either a `Quantity` angle or angular speed passed in to the argument `x`. [#5653]

- Added a new equivalence `molar_mass_amu` between g/mol to atomic mass units. [#6040, #6113]
- `Quantity` has gained a new `to_value` method which returns the value of the quantity in a given unit. [#6127]
- `Quantity` now supports the `@` operator for matrix multiplication that was introduced in Python 3.5, for all supported versions of numpy. [#6144]
- `Quantity` supports the new `__array_ufunc__` protocol introduced in numpy 1.13. As a result, operations that involve unit conversion will be sped up considerably (by up to a factor of two for costly operations such as trigonometric ones). [#2583]

astropy.utils

- Added a new `dataurl_mirror` configuration item in `astropy.utils.data` that is used to indicate a mirror for the astropy data server. [#5547]
- Added a new convenience method `get_cached_urls` to `astropy.utils.data` for getting a list of the URLs in your cache. [#6242]

astropy.wcs

- Upgraded the included wcslib to version 5.16. [#6225]

  The minimum required version of wcslib in is 5.14.

API Changes

astropy.analytic_functions

- This entire sub-package is deprecated because blackbody has been moved to `astropy.modeling.blackbody`. [#6191]

astropy.convolution

- Major change in convolution behavior and keyword arguments. `astropy.convolution.convolve_fft` replaced `interpolate_nan` with `nan_treatment`, and `astropy.convolution.convolve` received a new `nan_treatment` argument. `astropy.convolution.convolve` also no longer double-interpolates interpolates over NaNs, although that is now available as a separate `astropy.convolution.interpolate_replace_nans` function. See the backwards compatibility note for more on how to get the old behavior (and why you probably don’t want to.) [#5782]

astropy.coordinates

- The `astropy.coordinates.Galactic` frame previously was had the cartesian ordering ‘w’, ‘u’, ‘v’ (for ‘x’, ‘y’, and ‘z’, respectively). This was an error and against the common convention. The ‘x’, ‘y’, and ‘z’ axes now map to ‘u’, ‘v’, and ‘w’, following the right-handed (‘u’ points to the Galactic center) convention. [#6330]
- Removed deprecated `angles.rotation_matrix` and `angles.angle_axis`. Use the routines in `coordinates.matrix_utilities` instead. [#6170]
- `EarthLocation.latitude` and `EarthLocation.longitude` are now deprecated in favor of `EarthLocation.lat` and `EarthLocation.lon`. They former will be removed in a future version. [#6237]
- The `FrameAttribute` class and subclasses have been renamed to just contain `Attribute`. For example, `QuantityFrameAttribute` is now `QuantityAttribute`. [#6300]

astropy.cosmology

- Cosmological models do not include any contribution from neutrinos or photons by default – that is, the default value of Tcmb0 is 0. This does not affect built in models (such as WMAP or Planck). [#6112]

astropy.io.fits

- Remove deprecated `NumCode` and `ImgCode` properties on FITS `_ImageBaseHDU`. Use module-level constants `BITPIX2DTYPE` and `DTYPE2BITPIX` instead. [#4993]
- `comments` meta key (which is `io.ascii`’s table convention) is output to `COMMENT` instead of `COMMENTS` header. Similarly, `COMMENT` headers are read into `comments` meta [#6097]
- Remove compatibility code which forced loading all HDUs on close. The old behavior can be used with `lazy_load_hdus=False`. Because of this change, trying to access the `.data` attribute from an HDU which is not loaded now raises a `IndexError` instead of a `ValueError`. [#6082]
- Deprecated `clobber` keyword; use `overwrite`. [#6203]
- Add EXTVER column to the output of `HDUList.info()`. [#6124]

astropy.modeling

- Removed deprecated `Redshift` model; Use `RedshiftScaleFactor`. [#6053]
- Removed deprecated `Pix2Sky_AZP.check_mu` and `Pix2Sky_SZP.check_mu` methods. [#6170]
- Deprecated `GaussianAbsorption1D` model, as it can be better

represented by subtracting `Gaussian1D` from `Const1D`. [#6200]
- Added method `sum_of_implicit_terms` to `Model`, needed when performing a linear fit to a model that has built-in terms with no corresponding parameters (primarily the `1*x` term of `Shift`). [#6174]

astropy.nddata

- Removed deprecated usage of parameter `propagate_uncertainties` as a positional keyword. [#6170]
- Removed deprecated `support_correlated` attribute. [#6170]
- Removed deprecated `propagate_add`, `propagate_subtract`, `propagate_multiply` and `propagate_divide` methods. [#6170]

astropy.stats

- Removed the deprecated `sig` and `varfunc` keywords in the `sigma_clip` function. [#5715]
- Added `modify_sample_size` keyword to `biweight_midvariance` function. [#5991]

astropy.table

- In Python 3, when getting an item from a bytestring Column it is now converted to `str`. This means comparing a single item to a `bytes` object will always fail, and instead one must compare with a `str` object. [#5700]
- Removed the deprecated `data` property of Row. [#5729]
- Removed the deprecated functions `join`, `hstack`, `vstack` and `get_groups` from np_utils. [#5729]
- Added `name` parameter to method `astropy.table.Table.add_column` and `names` parameter to method `astropy.table.Table.add_columns`, to provide the flexibility to add unnamed columns, mixin objects and also to specify explicit names. Default names will be used if not specified. [#5996]
- Added optional `axis` parameter to `insert` method for `Column` and `MaskedColumn` classes. [#6092]

astropy.units

- Moved `units.cgs.emu` to `units.deprecated.emu` due to ambiguous definition of “emu”. [#4918, #5906]
- `jupiterMass`, `earthMass`, `jupiterRad`, and `earthRad` no longer have their prefixed units included in the standard units. If needed, they can still be found in `units.deprecated`. [#5661]

- `solLum`,``solMass``, and `solRad` no longer have their prefixed units included in the standard units. If needed, they can still be found in `units.required_by_vounit`, and are enabled by default. [#5661]
- Removed deprecated `Unit.get_converter`. [#6170]
- Internally, astropy replaced use of `.to(unit).value` with the new `to_value(unit)` method, since this is somewhat faster. Any subclasses that overwrote `.to`, should also overwrite `.to_value` (or possibly just the private `._to_value` method. (If you did this, please let us know what was lacking that made this necessary!). [#6137]

astropy.utils

- Removed the deprecated compatibility modules for Python 2.6 (`argparse`, `fractions`, `gzip`, `odict`, `subprocess`) [#5975,#6157,#6164]
- Removed the deprecated `zest.releaser` machinery. [#6282]

astropy.visualization

- Removed the deprecated `scale_image` function. [#6170]

astropy.vo

- Cone Search now issues deprecation warning because it is moved to Astroquery 0.3.5 and will be removed from Astropy in a future version. [#5558, #5904]
- The `astropy.vo.samp` package has been moved to `astropy.samp`, and no longer supports HTTPS/SSL. [#6201, #6213]

astropy.wcs

- Removed deprecated `wcs.rotateCD`. [#6170]

Bug Fixes

astropy.convolution

- Major change in convolution behavior and keyword arguments: `astropy.convolution.convolve` was not performing normalized convolution in earlier versions of astropy. [#5782]
- Direct convolution previously implemented the wrong definition of convolution. This error only affects *asymmetric* kernels. [#6267]

astropy.coordinates

- The `astropy.coordinates.Galactic` frame had an incorrect ording for

the ‘u’, ‘v’, and ‘w’ cartesian coordinates. [#6330]
- The `astropy.coordinates.search_around_sky`, `astropy.coordinates.search_around_3d`, and `SkyCoord` equivalent methods now correctly yield an `astropy.coordinates.Angle` as the third return type even if there are no matches (previously it returned a raw Quantity). [#6347]

astropy.io.ascii

- Fix an issue where the fast C-reader was dropping table comments for a table with no data lines. [#8274]

astropy.io.fits

- `comments` meta key (which is `io.ascii`’s table convention) is output to `COMMENT` instead of `COMMENTS` header. Similarly, `COMMENT` headers are read into `comments` meta [#6097]
- Use more sensible fix values for invalid NAXISj header values. [#5935]
- Close file on error to avoid creating a `ResourceWarning` warning about an unclosed file. [#6168, #6177]

astropy.modeling

- Creating a compound model where one of the submodels is a compound model whose parameters were changed now uses the updated parameters and not the parameters of the original model. [#5741]
- Allow `Mapping` and `Identity` to be fittable. [#6018]
- Gaussian models now impose positive `stddev` in fitting. [#6019]
- OrthoPolynomialBase (Chebyshev2D / Legendre2D) models were being evaluated incorrectly when part of a compound model (using the parameters from the original model), which in turn caused fitting to fail as a no-op. [#6085]
- Allow `Ring2D` to be defined using `r_out`. [#6192]
- Make `LinearLSQFitter` produce correct results with fixed model parameters and allow `Shift` and `Scale` to be fitted with `LinearLSQFitter` and `LevMarLSQFitter`. [#6174]

astropy.stats

- Allow to choose which median function is used in `mad_std` and `median_absolute_deviation`. And allow to use these functions with a multi-dimensional `axis`. [#5835]
- Fixed `biweight_midvariance` so that by default it returns a variance that agrees with the standard definition. [#5991]

astropy.table

- Fix a problem with vstack for bytes columns in Python 3. [#5628]
- Fix QTable add/insert row for multidimensional Quantity. [#6092]

astropy.time

- Fixed the initial condition of `TimeFITS` to allow scale, FITS scale and FITS realization to be checked and equated properly. [#6202]

astropy.visualization

- Fixed a bug that caused the default WCS to return coordinates offset by one. [#6339]

astropy.vo

- Fixed a bug in vo.samp when stopping a hub for which a lockfile was not created. [#6211]

Other Changes and Additions

- Numpy 1.7 and 1.8 are no longer supported. [#6006]
- Python 3.3 is no longer suppored. [#6020]
- The bundled ERFA was updated to version 1.4.0. [#6239]
- The bundled version of pytest has now been removed, but the astropy.tests.helper.pytest import will continue to work properly. Affiliated packages should nevertheless transition to importing pytest directly rather than from astropy.tests.helper. This also means that pytest is now a formal requirement for testing for both Astropy and for affiliated packages. [#5694]

1.3.3 (2017-05-29)

Bug Fixes

astropy.coordinates

- Fixed a bug where `StaticMatrixTransform` erroneously copied frame attributes from the input coordinate to the output frame. In practice, this didn’t actually affect any transforms in Astropy but may change behavior for users who explicitly used the `StaticMatrixTransform` in their own code. [#6045]
- Fixed `get_icrs_coordinates` to loop through all the urls in case one raises an exception. [#5864]

astropy.io.fits

- Fix table header not written out properly when `fits.writeto()` convenience function is used. [#6042]
- Fix writing out read-only arrays. [#6036]
- Extension headers are written out properly when the `fits.update()` convenience function is used. [#6058]
- Angstrom, erg, G, and barn are no more reported as deprecated FITS units. [#5929]

astropy.table

- Fix problem with Table pprint/pformat raising an exception for non-UTF-8 compliant bytestring data. [#6117]

astropy.units

- Allow strings 'nan' and 'inf' as Quantity inputs. [#5958]
- Add support for `positive` and `divmod` ufuncs (new in numpy 1.13). [#5998, #6020, #6116]

astropy.utils

- On systems that do not have `pkg_resources` non-numerical additions to version numbers like `dev` or `rc1` are stripped in `minversion` to avoid a `TypeError` in `distutils.version.LooseVersion` [#5944]
- Fix `auto_download` setting ignored in `Time.ut1`. [#6033]

astropy.visualization

- Fix bug in ManualInterval which caused the limits to be returned incorrectly if set to zero, and fix defaults for ManualInterval in the presence of NaNs. [#6088]
- Get rid of warnings that occurred when slicing a cube due to the tick locator trying to find ticks for the sliced axis. [#6104]
- Accept normal Matplotlib keyword arguments in set_xlabel and set_ylabel functions. [#5686, #5692, #6060]
- Fix a bug that caused labels to be missing from frames with labels that could change direction mid-axis, such as EllipticalFrame. Also ensure that empty tick labels do not cause any warnings. [#6063]

1.3.2 (2017-03-30)

Bug Fixes

astropy.coordinates

- Ensure that checking equivalance of `SkyCoord` objects works with non-scalar attributes [#5884, #5887]
- Ensure that transformation to frames with multi-dimensional attributes works as expected [#5890, #5897]
- Make sure all `BaseRepresentation` objects can be output as strings. [#5889, #5897]

astropy.units

- Add support for `heaviside` ufunc (new in numpy 1.13). [#5920]

astropy.utils

- Fix to allow the C-based _fast_iterparse() VOTable XML parser to relloc() its buffers instead of overflowing them. [#5824, #5869]

Other Changes and Additions

- File permissions are revised in the released source distribution. [#5912]

1.3.1 (2017-03-18)

New Features

astropy.utils

- The `deprecated_renamed_argument` decorator got a new `pending` parameter to suppress the deprecation warnings. [#5761]

Bug Fixes

astropy.coordinates

- Changed `SkyCoord` so that frame attributes which are not valid for the current `frame` (but are valid for other frames) are stored on the `SkyCoord` instance instead of the underlying `frame` instance (e.g., setting `relative_humidity` on an ICRS `SkyCoord` instance.) [#5750]
- Ensured that `position_angle` and `separation` give correct answers for frames with different equinox (see #5722). [#5762]

astropy.io.fits

- Fix problem with padding bytes written for BinTable columns converted from

unicode [#5280, #5287, #5288, #5296].
- Fix out-of-order TUNITn cards when writing tables to FITS. [#5720]
- Recognize PrimaryHDU when non boolean values are present for the 'GROUPS' header keyword. [#5808]
- Fix the insertion of new keywords in compressed image headers (`CompImageHeader`). [#5866]

astropy.modeling

- Fixed a problem with setting `bounding_box` on 1D models. [#5718]
- Fixed a broadcasting problem with weighted fitting of 2D models with `LevMarLSQFitter`. [#5788]
- Fixed a problem with passing kwargs to fitters, specifically `verblevel`. [#5815]
- Changed FittingWithOutlierRemoval to reject on the residual to the fit [#5831]

astropy.stats

- Fix the psd normalization for Lomb-Scargle periodograms in the presence of noise. [#5713]
- Fix bug in the autofrequency range when `minimum_frequency` is specified but `maximum_frequency` is not. [#5738]
- Ensure that a masked array is returned when sigma clipping fully masked data. [#5711]

astropy.table

- Fix problem where key for caching column format function was not sufficiently unique. [#5803]
- Handle sorting NaNs and masked values in jsviewer. [#4052, #5572]
- Ensure mixin columns can be added to a table using a scalar value for the right-hand side if the type supports broadcasting. E.g., for an existing `QTable`, `t['q'] = 3*u.m` will now add a column as expected. [#5820]
- Fixes the bug of setting/getting values from rows/columns of a table using numpy array scalars. [#5772]

astropy.units

- Fixed problem where IrreducibleUnits could fail to unpickle. [#5868]

astropy.utils

- Avoid importing `ipython` in `utils.console` until it is necessary, to prevent deprecation warnings when importing, e.g., `Column`. [#5755]

astropy.visualization

- Avoid importing matplotlib.pyplot when importing astropy.visualization.wcsaxes. [#5680, #5684]
- Ignore Numpy warnings that happen in coordinate transforms in WCSAxes. [#5792]
- Fix compatibility issues between WCSAxes and Matplotlib 2.x. [#5786]
- Fix a bug that caused WCSAxes frame visual properties to not be copied over when resetting the WCS. [#5791]

astropy.extern

- Fixed a bug where PLY was overwriting its generated files. [#5728]

Other Changes and Additions

- Fixed a deprecation warning that occurred when running tests with astropy.test(). [#5689]
- The deprecation of the `clobber` argument (originally deprecated in 1.3.0) in the `io.fits` write functions was changed to a "pending" deprecation (without displaying warnings) for now. [#5761]
- Updated bundled astropy-helpers to v1.3.1. [#5880]

1.3 (2016-12-22)

New Features

astropy.convolution

- The `convolve` and `convolve_fft` arguments now support a `mask` keyword, which allows them to also support `NDData` objects as inputs. [#5554]

astropy.coordinates

- Added an `of_address` classmethod to `EarthLocation` to enable fast creation of `EarthLocation` objects given an address by querying the Google maps API [#5154].
- A new routine, `get_body_barycentric_posvel` has been added that allows one to calculate positions as well as velocities for solar system bodies. For JPL kernels, this roughly doubles the execution time, so if one requires only the positions, one should use `get_body_barycentric`. [#5231]
- Transformations between coordinate systems can use the more accurate

JPL ephemerides. [#5273, #5436]

- Arithmetic on representations, such as addition of two representations, multiplication with a `Quantity`, or calculating the norm via `abs`, has now become possible. Furthermore, there are new methods `mean`, `sum`, `dot`, and `cross`. For all these, the representations are treated as vectors in cartesian space (temporarily converting to `CartesianRepresentation` if necessary). [#5301] has now become possible. Furthermore, there are news methods `mean`, `sum`, `dot`, and `cross` with obvious meaning. [#5301] multiplication with a `Quantity` has now become possible. Furthermore, there are new methods `norm`, `mean`, `sum`, `dot`, and `cross`. In all operations, the representations are treated as vectors. They are temporarily converted to `CartesianRepresentation` if necessary. [#5301]
- `CartesianRepresentation` can be initialized with plain arrays by passing in a `unit`. Furthermore, for input with a vector array, the coordinates no longer have to be in the first dimension, but can be at any `xyz_axis`. To complement the latter, a new `get_xyz(xyz_axis)` method allows one to get a vector array out along a given axis. [#5439]

astropy.io.ascii

- Files with “Fortran-style” columns (i.e. double-precision scientific notation with a character other than “e”, like `1.495978707D+13`) can now be parsed by the fast reader natively. [#5552]
- Allow round-tripping masked data tables in most formats by using an empty string `''` as the default representation of masked values when writing. [#5347]
- Allow reading HTML tables with unicode column values in Python 2.7. [#5410]
- Check for self-consistency of ECSV header column names. [#5463]
- Produce warnings when writing an IPAC table from an astropy table that contains metadata not supported by the IPAC format. [#4700]

astropy.io.fits

- “Lazy” loading of HDUs now occurs - when an HDU is requested, the file is only read up to the point where that HDU is found. This can mean a substantial speedup when accessing files that have many HDUs. [#5065]

astropy.io.misc

- Added `io.misc.yaml` module to support serializing core astropy objects using the YAML protocol. [#5486]

astropy.io.registry

- Added `delay_doc_updates` contextmanager to postpone the formatting of the documentation for the `read` and `write` methods of the class to optionally reduce the import time. [#5275]

astropy.modeling

- Added a class to combine astropy fitters and functions to remove outliers e. g., sigma clip. [#4760]
- Added a `Tabular` model. [#5105]
- Added `Hermite1D` and `Hermite2D` polynomial models [#5242]
- Added the injection of EntryPoints into astropy.modeling.fitting if they inherit from Fitters class. [#5241]
- Added bounding box to `Lorentz1D` and `MexicanHat1D` models. [#5393]
- Added `Planar2D` functional model. [#5456]
- Updated `Gaussian2D` to accept no arguments (will use default x/y_stddev and theta). [#5537]

astropy.nddata

- Added `keep` and `**kwargs` parameter to `support_nddata`. [#5477]

astropy.stats

- Added `axis` keyword to `biweight_location` and `biweight_midvariance`. [#5127, #5158]

astropy.table

- Allow renaming mixin columns. [#5469]
- Support generalized value formatting for mixin columns in tables. [#5274]
- Support persistence of table indices when pickling and copying table. [#5468]

astropy.tests

- Install both runtime and test dependencies when running the ./setup.py test command. These dependencies are specified by the install_requires and tests_require keywords via setuptools. [#5092]
- Enable easier subclassing of the TestRunner class. [#5505]

astropy.time

- `light_travel_time` can now use more accurate JPL ephemerides. [#5273, #5436]

astropy.units

- Added `pixel_scale` and `plate_scale` equivalencies. [#4987]
- The `spectral_density` equivalency now supports transformations of luminosity density. [#5151]
- `Quantity` now accepts strings consisting of a number and unit such as '10 km/s'. [#5245]

astropy.utils

- Added a new decorator: `deprecated_renamed_argument`. This can be used to rename a function argument, while it still allows for the use of the older argument name. [#5214]

astropy.visualization

- Added a `make_lupton_rgb` function to generate color images from three greyscale images, following the algorithm of Lupton et al. (2004). [#5535]
- Added `data` and `interval` inputs to the `ImageNormalize` class. [#5206]
- Added a new `simple_norm` convenience function. [#5206]
- Added a default stretch for the `Normalization` class. [#5206].
- Added a default `vmin/vmax` for the `ManualInterval` class. [#5206].
- The `wcsaxes` subpackage has now been integrated in astropy as `astropy.visualization.wcsaxes`. This allows plotting of astronomical data/coordinate systems in Matplotlib. [#5496]

astropy.wcs

- Improved `footprint_to_file`: allow to specify the coordinate system, and use by default the one from `RADESYS`. Overwrite the file instead of appending to it. [#5494]

API Changes

astropy.convolution

- `discretize_model` now raises an exception if non-integer ranges are used. Previously it had incorrect behavior but did not raise an exception. [#5538]

astropy.coordinates

- `SkyCoord`, `ICRS`, and other coordinate objects, as well as the underlying representations such as `SphericalRepresentation` and `CartesianRepresentation` can now be reshaped using methods named like the numpy ones for `ndarray` (`reshape`, `swapaxes`, etc.) [#4123, #5254, #5482]

- The `obsgeoloc` and `obsgeovel` attributes of `GCRS` and `PrecessedGeocentric` frames are now stored and returned as `CartesianRepresentation` objects, rather than `Quantity` objects. Similarly, `EarthLocation.get_gcrs_posvel` now returns a tuple of `CartesianRepresentation` objects. [#5253]
- `search_around_3d` and `search_around_sky` now return units for the distance matching their input argument when no match is found, instead of `dimensionless_unscaled`. [#5528]

astropy.io.ascii

- ASCII writers now accept an 'overwrite' argument. The default behavior is changed so that a warning will be issued when overwriting an existing file unless `overwrite=True`. In a future version this will be changed from a warning to an exception to prevent accidentally overwriting a file. [#5007]
- The default representation of masked values when writing tables was changed from `'--'` to the empty string `''`. Previously any user-supplied `fill_values` parameter would overwrite the class default, but now the values are prepended to the class default. [#5347]

astropy.io.fits

- The old `Header` interface, deprecated since Astropy 0.1 (PyFITS 3.1), has been removed entirely. See Header Interface Transition Guide for explanations on this change and help on the transition. [#5310]
- The following functions, classes and methods have been removed: `CardList`, `Card.key`, `Card.cardimage`, `Card.ascardimage`, `create_card`, `create_card_from_string`, `upper_key`, `Header.ascard`, `Header.rename_key`, `Header.get_history`, `Header.get_comment`, `Header.toTxtFile`, `Header.fromTxtFile`, `new_table`, `tdump`, `tcreate`, `BinTableHDU.tdump`, `BinTableHDU.tcreate`.
- Removed `txtfile` argument to the `Header` constructor.
- Removed usage of `Header.update` with `Header.update(keyword, value, comment)` arguments.
- Removed `startColumn` and `endColumn` arguments to the `FITS_record` constructor.
- The `clobber` argument in FITS writers has been renamed to `overwrite`. This change affects the following functions and methods: `tabledump`, `writeto`, `Header.tofile`, `Header.totextfile`,

`_BaseDiff.report`, `_BaseHDU.overwrite`, `BinTableHDU.dump` and `HDUList.writeto`. [#5171]
- Added an optional `copy` parameter to `fits.Header` which controls if a copy is made when creating an `Header` from another `Header`. [#5005, #5326]

astropy.io.registry

- `.fts` and `.fts.gz` files will be automatically identified as `io.fits` files if no explicit `format` is given. [#5211]
- Added an optional `readwrite` parameter for `get_formats` to filter formats for read or write. [#5275]

astropy.modeling

- `Gaussian2D` now raises an error if `theta` is set at the same time as `cov_matrix` (previously `theta` was silently ignored). [#5537]

astropy.table

- Setting an existing table column (e.g. `t['a'] = [1, 2, 3]`) now defaults to *replacing* the column with a column corresponding to the new value (using `t.replace_column()`) instead of doing an in-place update. Any existing meta-data in the column (e.g. the unit) is discarded. An in-place update is still done when the new value is not a valid column, e.g. `t['a'] = 0`. To force an in-place update use the pattern `t['a'][:] = [1, 2, 3]`. [#5556]
- Allow `collections.Mapping`-like `data` attribute when initializing a `Table` object (`dict`-like was already possible). [#5213]

astropy.tests

- The inputs to the `TestRunner.run_tests()` method now must be keyword arguments (no positional arguments). This applies to the `astropy.test()` function as well. [#5505]

astropy.utils

- Renamed `ignored` context manager in `compat.misc` to `suppress` to be consistent with https://bugs.python.org/issue19266 . [#5003]

astropy.visualization

- Deprecated the `scale_image` function. [#5206]
- The `mpl_normalize` module (containing the `ImageNormalize` class) is

now automatically imported with the `visualization` subpackage. [#5491]

astropy.vo

- The `clobber` argument in `VOSDatabase.to_json()` has been renamed to `overwrite`. [#5171]

astropy.wcs

- `wcs.rotateCD()` was deprecated without a replacement. [#5240]

Bug Fixes

astropy.coordinates

- Transformations between CIRS and AltAz now correctly account for the location of the observer. [#5591]
- GCRS frames representing a location on Earth with multiple obstimes are now allowed. This means that the solar system routines `get_body`, `get_moon` and `get_sun` now work with non-scalar times and a non-geocentric observer. [#5253]

astropy.io.ascii

- Fix issue with units or other astropy core classes stored in table meta. [#5605]

astropy.io.fits

- Copying a `fits.Header` using `copy` or `deepcopy` from the `copy` module will use `Header.copy` to ensure that modifying the copy will not alter the other original Header and vice-versa. [#4990, #5323]
- `HDUList.info()` no longer raises `AttributeError` in presence of `BZERO`. [#5508]
- Avoid exceptions with numpy 1.10 and up when using scaled integer data where `BZERO` has float type but integer value. [#4639, #5527]
- Converting a header card to a string now calls `self.verify('fix+warn')` instead of `self.verify('fix')` so headers with invalid keywords will not raise a `VerifyError` on printing. [#887,#5054]
- `FITS_Record._convert_ascii` now converts blank fields to 0 when a non-blank null column value is set. [#5134, #5394]

astropy.io.registry

- `read` now correctly raises an IOError if a file with an unknown extension can't be found, instead of raising IORegistryError: "Format could not be identified." [#4779]

astropy.time

- Ensure `Time` instances holding a single `delta_ut1_utc` can be copied, flattened, etc. [#5225]

astropy.units

- Operations involving `Angle` or `Distance`, or any other `SpecificTypeQuantity` instance, now also keep return an instance of the same type if the instance was the second argument (if the resulting unit is consistent with the specific type). [#5327]
- Inplace operations on `Angle` and `Distance` instances now raise an exception if the final unit is not equivalent to radian and meter, resp. Similarly, views as `Angle` and `Distance` can now only be taken from quantities with appropriate units, and views as `Quantity` can only be taken from logarithmic quanties such as `Magnitude` if the physical unit is dimensionless. [#5070]
- Conversion from quantities to logarithmic units now correctly causes a logarithmic quantity such as `Magnitude` to be returned. [#5183]

astropy.wcs

- SIP distortion for an alternate WCS is correctly initialized now by looking at the "CTYPE" values matching the alternate WCS. [#5443]

Other Changes and Additions

- The bundled ERFA was updated to version 1.3.0. This includes the leap second planned for 2016 Dec 31.

astropy.coordinates

- Initialization of `Angle` has been sped up for `Quantity` and `Angle` input. [#4970]
- The use of `np.matrix` instances in the transformations has been deprecated, since this class does not allow stacks of matrices. As a result, the semi-public functions `angles.rotation_matrix` and `angles.angle_axis` are also deprecated, in favour of the new routines with the same name in `coordinates.matrix_utilities`. [#5104]
- A new `BaseCoordinateFrame.cache` dictionary has been created to

expose the internal cache. This is useful when modifying representation data in-place without using `realize_frame`. Additionally, documentation for in-place operations on coordinates were added. [#5575]
- Coordinates and their representations are printed with a slightly different format, following how numpy >= 1.12 prints structured arrays. [#5423]

astropy.cosmology

- The default cosmological model has been changed to Planck 2015, and the citation strings have been updated. [#5372]

astropy.extern

- Updated the bundled `six` module to version 1.10.0. [#5521]
- Updated the astropy shipped version of `PLY` to version 3.9. [#5526]
- Updated the astropy shipped version of jQuery to v3.3.1, and dataTables to v1.10.12. [#5564]

astropy.io.fits

- Performance improvements for tables with many columns. [#4985]
- Removed obsolete code that was previously needed to properly implement the append mode. [#4793]

astropy.io.registry

- Reduced the time spent in the `get_formats` function. This also reduces the time it takes to import astropy subpackages, i.e. `astropy.coordinates`. [#5262]

astropy.units

- The functions `add_enabled_units`, `set_enabled_equivalencies` and `add_enabled_equivalencies` have been sped up by copying the current `_UnitRegistry` instead of building it from scratch. [#5306]
- To build the documentation, the `build_sphinx` command has been deprecated in favor of `build_docs`. [#5179]
- The `--remote-data` option to `python setup.py test` can now take different arguments: `--remote-data=none` is the same as not specifying `--remote-data` (skip all tests that require the internet), `--remote-data=astropy` skips all tests that need remote data except those that require only data from data.astropy.org, and `--remote-data=any` is the same as `--remote-data` (run all tests that use remote data). [#5506]
- The pytest `recwarn` fixture has been removed from the tests in favor of

`utils.catch_warnings`. [#5489]

- Deprecated escape sequences in strings (Python 3.6) have been removed. [#5489]

1.2.2 (2016-12-22)

Bug Fixes

astropy.io.ascii

- Fix a bug where the `fill_values` parameter was ignored when writing a table to HTML format. [#5379]

astropy.io.fits

- Handle unicode FITS BinTable column names on Python 2 [#5204, #4805]
- Fix reading of float values from ASCII tables, that could be read as float32 instead of float64 (with the E and F formats). These values are now always read as float64. [#5362]
- Fixed memoryleak when using the compression module. [#5399, #5464]
- Able to insert and remove lower case HIERARCH keywords in a consistent manner [#5313, #5321]

astropy.stats

- Fixed broadcasting in `sigma_clip` when using negative `axis`. [#4988]

astropy.table

- Assigning a logarithmic unit to a `QTable` column that did not have a unit yet now correctly turns it into the appropriate function quantity subclass (such as `Magnitude` or `Dex`). [#5345]
- Fix default value for `show_row_index` in `Table.show_in_browser`. [#5562]

astropy.units

- For inverse trig functions that operate on quantities, catch any warnings that occur from evaluating the function on the unscaled quantity value between __array_prepare__ and __array_wrap__. [#5153]
- Ensure `!=` also works for function units such as `MagUnit` [#5345]

astropy.wcs

- Fix use of the `relax` keyword in `to_header` when used to change the

output precision. [#5164]
- `wcs.to_header(relax=True)` adds a “-SIP” suffix to `CTYPE` when SIP distortion is present in the WCS object. [#5239]
- Improved log messages in `to_header`. [#5239]

Other Changes and Additions

- The bundled ERFA was updated to version 1.3.0. This includes the leap second planned for 2016 Dec 31.

astropy.stats

- `poisson_conf_interval` with `'kraft-burrows-nousek'` interval is now faster and useable with SciPy versions < 0.14. [#5064, #5290]

1.2.1 (2016-06-22)

Bug Fixes

astropy.io.fits

- Fixed a bug that caused TFIELDS to not be in the correct position in compressed image HDU headers under certain circumstances, which created invalid FITS files. [#5118, #5125]

astropy.units

- Fixed an `ImportError` that occurred whenever `astropy.constants` was imported before `astropy.units`. [#5030, #5121]
- Magnitude zero points used to define `STmag`, `ABmag`, `M_bol` and `m_bol` are now collected in `astropy.units.magnitude_zero_points`. They are not enabled as regular units by default, but can be included using `astropy.units.magnitude_zero_points.enable()`. This makes it possible to round-trip magnitudes as originally intended. [#5030]

1.2 (2016-06-19)

General

- Astropy now requires Numpy 1.7.0 or later. [#4784]

New Features

astropy.constants

- Add `L_bol0`, the luminosity corresponding to absolute bolometric magnitude zero. [#4262]

astropy.coordinates

- `CartesianRepresentation` now includes a transform() method that can take a 3x3 matrix to transform coordinates. [#4860]
- Solar system and lunar ephemerides accessible via `get_body`, `get_body_barycentric` and `get_moon` functions. [#4890]
- Added astrometric frames (i.e., a frame centered on a particular point/object specified in another frame). [#4909, #4941]
- Added `SkyCoord.spherical_offsets_to` method. [#4338]
- Recent Earth rotation (IERS) data are now auto-downloaded so that AltAz transformations for future dates now use the most accurate available rotation values. [#4436]
- Add support for heliocentric coordinate frames. [#4314]

astropy.cosmology

- `angular_diameter_distance_z1z2` now supports the computation of the angular diameter distance between a scalar and an array like argument. [#4593] The method now supports models with negative Omega_k0 (positive curvature universes) [#4661] and allows z2 < z1.

astropy.io.ascii

- File name could be passed as `Path` object. [#4606]
- Check that columns in `formats` specifier exist in the output table when writing. [#4508, #4511]
- Allow trailing whitespace in the IPAC header lines. [#4758]
- Updated to filter out the default parser warning of BeautifulSoup. [#4551]
- Added support for reading and writing reStructuredText simple tables. [#4812]

astropy.io.fits

- File name could be passed as `Path` object. [#4606]
- Header allows a dictionary-like cards argument during creation. [#4663]
- New function `convenience.table_to_hdu` to allow creating a FITS HDU object directly from an astropy `Table`. [#4778]
- New optional arguments `ignore_missing` and `remove_all` are added to `astropy.io.fits.header.remove()`. [#5020]

astropy.io.registry

- Added custom `IORegistryError`. [#4833]

astropy.io.votable

- File name could be passed as `Path` object. [#4606]

astropy.modeling

- Added the fittable=True attribute to the Scale and Shift models with tests. [#4718]
- Added example plots to docstrings for some build-in models. [#4008]

astropy.nddata

- `UnknownUncertainty` new subclass of `NDUncertainty` that can be used to save uncertainties that cannot be used for error propagation. [#4272]
- `NDArithmeticMixin`: `add`, `subtract`, `multiply` and `divide` can be used as classmethods but require that two operands are given. These operands don't need to be NDData instances but they must be convertible to NDData. This conversion is done internally. Using it on the instance does not require (but also allows) two operands. [#4272, #4851]
- `NDDataRef` new subclass that implements `NDData` together with all currently available mixins. This class does not implement additional attributes, methods or a numpy.ndarray-like interface like `NDDataArray`. attributes, methods or a numpy.ndarray-like interface like `NDDataArray`. [#4797]

astropy.stats

- Added `axis` keyword for `mad_std` function. [#4688, #4689]
- Added Bayesian and Akaike Information Criteria. [#4716]
- Added Bayesian upper limits for Poisson count rates. [#4622]
- Added `circstats`; a module for computing circular statistics. [#3705, #4472]
- Added `jackknife` resampling method. [#3708, #4439]
- Updated `bootstrap` to allow bootstrapping statistics with multiple outputs. [#3601]
- Added `LombScargle` class to compute Lomb-Scargle periodograms [#4811]

astropy.table

- `Table.show_in_notebook` and

`Table.show_in_browser(jsviewer=True)` now yield tables with an "idx" column, allowing easy identification of the index of a row even when the table is re-sorted in the browser. [#4404]
- Added `AttributeError` when trying to set mask on non-masked table. [#4637]
- Allow to use a tuple of keys in `Table.sort`. [#4671]
- Added `itercols`; a way to iterate through columns of a table. [#3805, #4888]
- `Table.show_in_notebook` and the default notebook display (i.e., `Table._repr_html_`) now use consistent table styles which can be set using the `astropy.table.default_notebook_table_class` configuration item. [#4886]
- Added interface to create `Table` directly from any table-like object that has an `__astropy_table__` method. [#4885]

astropy.tests

- Enable test runner to obtain documentation source files from directory other than "docs". [#4748]

astropy.time

- Added caching of scale and format transformations for improved performance. [#4422]
- Recent Earth rotation (IERS) data are now auto-downloaded so that UT1 transformations for future times now work out of the box. [#4436]
- Add support for barycentric/heliocentric time corrections. [#4314]

astropy.units

- The option to use tuples to indicate fractional powers of units, deprecated in 0.3.1, has been removed. [#4449]
- Added slug to imperial units. [#4670]
- Added Earth radius (`R_earth`) and Jupiter radius (`R_jup`) to units. [#4818]
- Added a `represents` property to allow access to the definition of a named unit (e.g., `u.kpc.represents` yields `1000 pc`). [#4806]
- Add bolometric absolute and apparent magnitudes, `M_bol` and `m_bol`. [#4262]

astropy.utils

- `Path` object could be passed to `get_readable_fileobj`. [#4606]
- Implemented a generic and extensible way of merging metadata. [#4459]

- Added `format_doc` decorator which allows to replace and/or format the current docstring of an object. [#4242]
- Added a new context manager `set_locale` to temporarily set the current locale. [#4363]
- Added new IERS_Auto class to auto-download recent IERS (Earth rotation) data when required by coordinate or time transformations. [#4436]

astropy.visualization

- Add zscale interval based on Numdisplay's implementation. [#4776]

API changes

astropy.config

- The deprecated `ConfigurationItem` and `ConfigAlias` classes and the `save_config`, `get_config_items`, and `generate_all_config_items` functions have now been removed. [#2767, #4446]

astropy.coordinates

- Removed compatibility layer for pre-v0.4 API. [#4447]
- Added `copy` keyword-only argument to allow initialization without copying the (possibly large) input coordinate arrays. [#4883]

astropy.cosmology

- Improve documentation of z validity range of cosmology objects [#4882, #4949]

astropy.io.ascii

- Add a way to control HTML escaping when writing a table as an HTML file. [#4423]

astropy.io.fits

- Two optional boolean arguments `ignore_missing` and `remove_all` are added to `Header.remove`. [#5020]

astropy.modeling

- Renamed `Redshift` model to `RedshiftScaleFactor`. [#3672]
- Inputs (`coords` and `out`) to `render` function in `Model` are converted to float. [#4697]

- `RotateNative2Celestial` and `RotateCelestial2Native` are now implemented as subclasses of `EulerAngleRotation`. [#4881, #4940]

astropy.nddata

- `NDDataBase` does not set the private uncertainty property anymore. This only affects you if you subclass `NDDataBase` directly. [#4270]
- `NDDataBase`: the `uncertainty`-setter is removed. A similar one is added in `NDData` so this also only affects you if you subclassed `NDDataBase` directly. [#4270]
- `NDDataBase`: `uncertainty`-getter returns `None` instead of the private uncertainty and is now abstract. This getter is moved to `NDData` so it only affects direct subclasses of `NDDataBase`. [#4270]
- `NDData` accepts a Quantity-like data and an explicitly given unit. Before a ValueError was raised in this case. The final instance will use the explicitly given unit-attribute but doesn't check if the units are convertible and the data will not be scaled. [#4270]
- `NDData` : the given mask, explicit or implicit if the data was masked, will be saved by the setter. It will not be saved directly as the private attribute. [#4879]
- `NDData` accepts an additional argument `copy` which will copy every parameter before it is saved as attribute of the instance. [#4270]
- `NDData`: added an `uncertainty.getter` that returns the private attribute. It is equivalent to the old `NDDataBase.uncertainty`-getter. [#4270]
- `NDData`: added an `uncertainty.setter`. It is slightly modified with respect to the old `NDDataBase.uncertainty`-setter. The changes include:
- if the uncertainty has no uncertainty_type an info message is printed instead of a TypeError and the uncertainty is saved as `UnknownUncertainty` except the uncertainty is None. [#4270]
- the requirement that the uncertainty_type of the uncertainty needs to be a string was removed. [#4270]
- if the uncertainty is a subclass of NDUncertainty the parent_nddata attribute will be set so the uncertainty knows to which data it belongs. This is also a Bugfix. [#4152, #4270]
- `NDData`: added a `meta`-getter, which will set and return an empty OrderedDict if no meta was previously set. [#4509, #4469]
- `NDData`: added an `meta`-setter. It requires that the meta is dictionary-like (it also accepts Headers or ordered dictionaries and others) or None. [#4509, #4469, #4921]
- `NDArithmeticMixin`: The operand in arithmetic methods (`add`, ...)

doesn't need to be a subclass of `NDData`. It is sufficient if it can be converted to one. This conversion is done internally. [#4272]

- `NDArithmeticMixin`: The arithmetic methods allow several new arguments to control how or if different attributes of the class will be processed during the operation. [#4272]
- `NDArithmeticMixin`: Giving the parameter `propagate_uncertainties` as positional keyword is deprecated and will be removed in the future. You now need to specify it as keyword-parameter. Besides `True` and `False` also `None` is now a valid value for this parameter. [#4272, #4851]
- `NDArithmeticMixin`: The wcs attribute of the operands is not compared and thus raises no ValueError if they differ, except if a `compare_wcs` parameter is specified. [#4272]
- `NDArithmeticMixin`: The arithmetic operation was split from a general `_arithmetic` method to different specialized private methods to allow subclasses more control on how the attributes are processed without overriding `_arithmetic`. The `_arithmetic` method is now used to call these other methods. [#4272]
- `NDSlicingMixin`: If the attempt at slicing the mask, wcs or uncertainty fails with a `TypeError` a Warning is issued instead of the TypeError. [#4271]
- `NDUncertainty`: `support_correlated` attribute is deprecated in favor of `supports_correlated` which is a property. Also affects `StdDevUncertainty`. [#4272]
- `NDUncertainty`: added the `__init__` that was previously implemented in `StdDevUncertainty` and takes an additional `unit` parameter. [#4272]
- `NDUncertainty`: added a `unit` property without setter that returns the set unit or if not set the unit of the parent. [#4272]
- `NDUncertainty`: included a `parent_nddata` property similar to the one previously implemented in StdDevUncertainty. [#4272]
- `NDUncertainty`: added an `array` property with setter. The setter will convert the value to a plain numpy array if it is a list or a subclass of a numpy array. [#4272]
- `NDUncertainty`: `propagate_multiply` and similar were removed. Before they were abstract properties and replaced by methods with the same name but with a leading underscore. The entry point for propagation is a method called `propagate`. [#4272]
- `NDUncertainty` and subclasses: implement a representation (`__repr__`). [#4787]
- `StdDevUncertainty`: error propagation allows an explicitly given

correlation factor, which may be a scalar or an array which will be taken into account during propagation. This correlation must be determined manually and is not done by the uncertainty! [#4272]

- `StdDevUncertainty`: the `array` is converted to a plain numpy array only if it's a list or a subclass of numpy.ndarray. Previously it was always cast to a numpy array but also allowed subclasses. [#4272]
- `StdDevUncertainty`: setting the `parent_nddata` does not compare if the shape of it's array is identical to the parents data shape. [#4272]
- `StdDevUncertainty`: the `array.setter` doesn't compare if the array has the same shape as the parents data. [#4272]
- `StdDevUncertainty`: deprecated `support_correlated` in favor of `supports_correlated`. [#4272, #4828]
- `StdDevUncertainty`: deprecated `propagate_add` and similar methods in favor of `propagate`. [#4272, #4828]
- Allow `data` to be a named argument in `NDDataArray`. [#4626]

astropy.table

- `operations.unique` now has a `keep` parameter, which allows one to select whether to keep the first or last row in a set of duplicate rows, or to remove all rows that are duplicates. [#4632]
- `QTable` now behaves more consistently by making columns act as a `Quantity` even if they are assigned a unit after the table is created. [#4497, #4884]

astropy.units

- Remove deprecated `register` argument for Unit classes. [#4448]

astropy.utils

- The astropy.utils.compat.argparse module has now been deprecated. Use the Python 'argparse' module directly instead. [#4462]
- The astropy.utils.compat.odict module has now been deprecated. Use the Python 'collections' module directly instead. [#4466]
- The astropy.utils.compat.gzip module has now been deprecated. Use the Python 'gzip' module directly instead. [#4464]
- The deprecated `ScienceStateAlias` class has been removed. [#2767, #4446]
- The astropy.utils.compat.subprocess module has now been deprecated. Use the Python 'subprocess' module instead. [#4483]
- The astropy.utils.xml.unescaper module now also unescapes `'%2F'` to `'/'` and `'&&'` to `'&'` in a given URL. [#4699]
- The astropy.utils.metadata.MetaData descriptor has now two optional

parameters: doc and copy. [#4921]
- The default IERS (Earth rotation) data now is now auto-downloaded via a new class IERS_Auto. When extrapolating UT1-UTC or polar motion values outside the available time range, the values are now clipped at the last available value instead of being linearly extrapolated. [#4436]

astropy.wcs

- WCS objects can now be initialized with an ImageHDU or PrimaryHDU object. [#4493, #4505]
- astropy.wcs now issues an INFO message when the header has SIP coefficients but “-SIP” is missing from CTYPE. [#4814]

Bug fixes

astropy.coordinates

- Ameliorate a problem with `get_sun` not round-tripping due to approximations in the light deflection calculation. [#4952]
- Ensure that `angle_utilities.position_angle` accepts floats, as stated in the docstring. [#3800]
- Ensured that transformations for `GCRS` frames are correct for non-geocentric observers. [#4986]
- Fixed a problem with the `Quantity._repr_latex_` method causing errors when showing an `EarthLocation` in a Jupyter notebook. [#4542, #5068]

astropy.io.ascii

- Fix a problem where the fast reader (with use_fast_converter=False) can fail on non-US locales. [#4363]
- Fix astropy.io.ascii.read handling of units for IPAC formatted files. Columns with no unit are treated as unitless not dimensionless. [#4867, #4947]
- Fix problems the header parsing in the sextractor reader. [#4603, #4910]

astropy.io.fits

- `GroupsHDU.is_image` property is now set to `False`. [#4742]
- Ensure scaling keywords are removed from header when unsigned integer data is converted to signed type. [#4974, #5053]
- Made TFORMx keyword check more flexible in test of compressed images to enable compatibility of the test with cfitsio 3.380. [#4646, #4653]

astropy.io.votable

- The astropy.io.votable.validator.html module is updated to handle division by

zero when generating validation report. [#4699]
- KeyError when converting Table v1.2 numeric arrays fixed. [#4782]

astropy.modeling

- Refactored `AiryDisk2D`, `Sersic1D`, and `Sersic2D` models to be able to combine them as classes as well as instances. [#4720]
- Modified the “LevMarLSQFitter” class to use the weights in the calculation of the Jacobian. [#4751]

astropy.nddata

- `NDData` giving masked_Quantities as data-argument will use the implicitly passed mask, unit and value. [#4270]
- `NDData` using a subclass implementing `NDData` with `NDArithmeticMixin` now allows error propagation. [#4270]
- Fixed memory leak that happened when uncertainty of `NDDataArray` was set. [#4825, #4862]
- `StdDevUncertainty`: During error propagation the unit of the uncertainty is taken into account. [#4272]
- `NDArithmeticMixin`: `divide` and `multiply` yield correct uncertainties if only one uncertainty is set. [#4152, #4272]

astropy.stats

- Fix `sigma_clipped_stats` to use the `axis` argument. [#4726, #4808]

astropy.table

- Fixed bug where Tables created from existing Table objects were not inheriting the `primary_key` attribute. [#4672, #4930]
- Provide more detail in the error message when reading a table fails due to a problem converting column string values. [#4759]

astropy.units

- Exponentiation using a `Quantity` with a unit equivalent to dimensionless as base and an `array`-like exponent yields the correct result. [#4770]
- Ensured that with `spectral_density` equivalency one could also convert between `photlam` and `STmag`/`ABmag`. [#5017]

astropy.utils

- The astropy.utils.compat.fractions module has now been deprecated. Use the Python ‘fractions’ module directly instead. [#4463]

- Added `format_doc` decorator which allows to replace and/or format the current docstring of an object. [#4242]
- Attributes using the astropy.utils.metadata.MetaData descriptor are now included in the sphinx documentation. [#4921]

astropy.vo

- Relaxed expected accuracy of Cone Search prediction test to reduce spurious failures. [#4382]

astropy.wcs

- astropy.wcs.to_header removes “-SIP” from CTYPE when SIP coefficients are not written out, i.e. `relax` is either `False` or `None`. astropy.wcs.to_header appends “-SIP” to CTYPE when SIP coefficients are written out, i.e. `relax=True`. [#4814]
- Made `wcs.bounds_check` call `wcsprm_python2c`, which means it works even if `wcs.set` has not been called yet. [#4957, #4966].
- WCS objects can no longer be reverse-indexed, which was technically permitted but incorrectly implemented previously [#4962]

Other Changes and Additions

- Python 2.6 is no longer supported. [#4486]
- The bundled version of py.test has been updated to 2.8.3. [#4349]
- Reduce Astropy’s import time (`import astropy`) by almost a factor 2. [#4649]
- Cython prerequisite for building changed to v0.19 in install.rst [#4705, #4710, #4719]
- All astropy.modeling functionality that was deprecated in Astropy 1.0 has been removed. [#4857]
- Added instructions for installing Astropy into CASA. [#4840]
- Added an example gallery to the docs demonstrating short snippets/examples. [#4734]

1.1.2 (2016-03-10)

New Features

astropy.wcs

- The `astropy.wcs` module now exposes `WCSHDO_P*` constants that can be used to allow more control over output precision when using the `relax` keyword argument. [#4616]

Bug Fixes

astropy.io.ascii

- Fixed handling of CDS data file when no description is given and also included stripping out of markup for missing value from description. [#4437, #4474]

astropy.io.fits

- Fixed possible segfault during error handling in FITS tile compression. [#4489]
- Fixed crash on pickling of binary table columns with the 'X', 'P', or 'Q' format. [#4514]
- Fixed memory / reference leak that could occur when copying a `FITS_rec` object (the `.data` for table HDUs). [#520]
- Fixed a memory / reference leak in `FITS_rec` that occurred in a wide range of cases, especially after writing FITS tables to a file, but in other cases as well. [#4539]

astropy.modeling

- Fix a bug to allow instantiation of a modeling class having a parameter with a custom setter that takes two parameters `(value, model)` [#4656]

astropy.table

- Fixed bug when replacing a table column with a mixin column like Quantity or Time. [#4601]
- Disable initial ordering in jsviewer (`show_in_browser`, `show_in_notebook`) to respect the order from the Table. [#4628]

astropy.units

- Fixed sphinx issues on plotting quantities. [#4527]

astropy.utils

- Fixed latex representation of function units. [#4563]
- The `zest.releaser` hooks included in Astropy are now injected locally to Astropy, rather than being global. [#4650]

astropy.visualization

- Fixed `fits2bitmap` script to allow ext flag to contain extension names or numbers. [#4468]

- Fixed `fits2bitmap` default output filename generation for compressed FITS files. [#4468]
- Fixed `quantity_support` to ensure its conversion returns ndarray instances (needed for numpy >=1.10). [#4654]

astropy.wcs

- Fixed possible exception in handling of SIP headers that was introduced in v1.1.1. [#4492]
- Fixed a bug that caused WCS objects with a high dynamic range of values for certain parameters to lose precision when converted to a header. This occurred for example in cases of spectral cubes, where a spectral axis in Hz might have a CRVAL3 value greater than 1e10 but the spatial coordinates would have CRVAL1/2 values 8 to 10 orders of magnitude smaller. This bug was present in Astropy 1.1 and 1.1.1 but not 1.0.x. This has now been fixed by ensuring that all WCS keywords are output with 14 significant figures by default. [#4616]

Other Changes and Additions

- Updated bundled astropy-helpers to v1.1.2. [#4678]
- Updated bundled copy of WCSLIB to 5.14. [#4579]

1.1.1 (2016-01-08)

New Features

astropy.io.registry

- Allow `pathlib.Path` objects (available in Python 3.4 and later) for specifying the file name in registry read / write functions. [#4405]

astropy.utils

- `console.human_file_size` now accepts quantities with byte-equivalent units [#4373]

Bug Fixes

astropy.analytic_functions

- Fixed the blackbody functions' handling of overflows on some platforms (Windows with MSVC, older Linux versions) with a buggy `expm1` function. [#4393]

astropy.io.fits

- Fixed an bug where updates to string columns in FITS tables were not saved on Python 3. [#4452]

Other Changes and Additions

- Updated bundled astropy-helpers to v1.1.1. [#4413]

1.1 (2015-12-11)

New Features

astropy.config

- Added new tools `set_temp_config` and `set_temp_cache` which can be used either as function decorators or context managers to temporarily use alternative directories in which to read/write the Astropy config files and download caches respectively. This is especially useful for testing, though `set_temp_cache` may also be used as a way to provide an alternative (application specific) download cache for large data files, rather than relying on the default cache location in users' home directories. [#3975]

astropy.constants

- Added the Thomson scattering cross-section. [#3839]

astropy.convolution

- Added Moffat2DKernel. [#3965]

astropy.coordinates

- Added `get_constellation` function and `SkyCoord.get_constellation` convenience method to determine the constellation that a coordinate is in. [#3758]
- Added `PrecessedGeocentric` frame, which is based on GCRS, but precessed to a specific requested mean equinox. [#3758]
- Added `Supergalactic` frame to support de Vaucouleurs supergalactic coordinates. [#3892]
- `SphericalRepresentation` now has a `._unit_representation` class attribute to specify an equivalent UnitSphericalRepresentation. This allows subclasses of representations to pair up correctly. [#3757]
- Added functionality to support getting the locations of observatories by name. See `astropy.coordinates.EarthLocation.of_site`. [#4042]

- Added ecliptic coordinates, including `GeocentricTrueEcliptic`, `BarycentricTrueEcliptic`, and `HeliocentricTrueEcliptic`. [#3749]

astropy.cosmology

- Add Planck 2015 cosmology [#3476]
- Distance calculations now > 20-40x faster for the supplied cosmologies due to implementing Cython scalar versions of `FLRW.inv_efunc`.[#4127]

astropy.io.ascii

- Automatically use `guess=False` when reading if the file `format` is provided and the format parameters are uniquely specified. This update also removes duplicate format guesses to improve performance. [#3418]
- Calls to ascii.read() for fixed-width tables may now omit one of the keyword arguments `col_starts` or `col_ends`. Columns will be assumed to begin and end immediately adjacent to each other. [#3657]
- Add a function `get_read_trace()` that returns a traceback of the attempted read formats for the last call to `astropy.io.ascii.read`. [#3688]
- Supports LZMA decompression via `get_readable_fileobj` [#3667]
- Allow `-` character is Sextractor format column names. [#4168]
- Improve DAOphot reader to read multi-aperture files [#3535, #4207]

astropy.io.fits

- Support reading and writing from bzip2 compressed files. i.e. `.fits.bz2` files. [#3789]
- Included a new command-line script called `fitsinfo` to display a summary of the HDUs in one or more FITS files. [#3677]

astropy.io.misc

- Support saving all meta information, description and units of tables and columns in HDF5 files [#4103]

astropy.io.votable

- A new method was added to `astropy.io.votable.VOTable`, `get_info_by_id` to conveniently find an `INFO` element by its `ID` attribute. [#3633]
- Instances in the votable tree now have better `__repr__` methods. [#3639]

astropy.logger.py

- Added log levels (e.g., DEBUG, INFO, CRITICAL) to `astropy.log` [#3947]

astropy.modeling

- Added a new `Parameter.validator` interface for setting a validation method on individual model parameters. See the `Parameter` documentation for more details. [#3910]
- The projection classes that are named based on the 3-letter FITS WCS projections (e.g. `Pix2Sky_TAN`) now have aliases using longer, more descriptive names (e.g. `Pix2Sky_Gnomonic`). [#3583]
- All of the standard FITS WCS projection types have been implemented in `astropy.modeling.projections` (by wrapping WCSLIB). [#3906]
- Added `Sersic1D` and `Sersic2D` model classes. [#3889]
- Added the Voigt profile to existing models. [#3901]
- Added `bounding_box` property and `render_model` function [#3909]

astropy.nddata

- Added `block_reduce` and `block_replicate` functions. [#3453]
- `extract_array` now offers different options to deal with array boundaries [#3727]
- Added a new `Cutout2D` class to create postage stamp image cutouts with optional WCS propagation. [#3823]

astropy.stats

- Added `sigma_lower` and `sigma_upper` keywords to `sigma_clip` to allow for non-symmetric clipping. [#3595]
- Added `cenfunc`, `stdfunc`, and `axis` keywords to `sigma_clipped_stats`. [#3792]
- `sigma_clip` automatically masks invalid input values (NaNs, Infs) before performing the clipping [#4051]
- Added the `histogram` routine, which is similar to `np.histogram` but includes several additional options for automatic determination of optimal histogram bins. Associated helper routines include `bayesian_blocks`, `friedman_bin_width`, `scott_bin_width`, and `knuth_bin_width`. This functionality was ported from the astroML library. [#3756]
- Added the `bayesian_blocks` routine, which implements a dynamic algorithm for locating change-points in various time series. [#3756]
- A new function `poisson_conf_interval()` was added to allow easy calculation of several standard formulae for the error bars on the mean of a Poisson variable estimated from a single sample.

astropy.table

- `add_column()` and `add_columns()` now have `rename_duplicate` option to rename new column(s) rather than raise exception when its name already exists. [#3592]
- Added `Table.to_pandas` and `Table.from_pandas` for converting to/from pandas dataframes. [#3504]
- Initializing a `Table` with `Column` objects no longer requires that the column `name` attribute be defined. [#3781]
- Added an `info` property to `Table` objects which provides configurable summary information about the table and its columns. [#3731]
- Added an `info` property to column classes (`Column` or mixins). This serves a dual function of providing configurable summary information about the column, and acting as a manager of column attributes such as name, format, or description. [#3731]
- Updated table and column representation to use the `dtype_info_name` function for the dtype value. Removed the default “masked=False” from the table representation. [#3868, #3869]
- Updated row representation to be consistent with the corresponding table representation for that row. Added HTML representation so a row displays nicely in IPython notebook.
- Added a new table indexing engine allowing for the creation of indices on one or more columns of a table using `add_index`. These indices enable new functionality such as searching for rows by value using `loc` and `iloc`, as well as increased performance for certain operations. [#3915, #4202]
- Added capability to include a structured array or recarray in a table as a mixin column. This allows for an approximation of nested tables. [#3925]
- Added `keep_byteorder` option to `Table.as_array()`. See the “API Changes” section below. [#4080]
- Added a new method `Table.replace_column()` to replace an existing column with a new data column. [#4090]
- Added a `tableclass` option to `Table.pformat()` to allow specifying a list of CSS classes added to the HTML table. [#4131]
- New CSS for jsviewer table [#2917, #2982, #4174]
- Added a new `Table.show_in_notebook` method that shows an interactive view of a Table (similar to `Table.show_in_browser(jsviewer=True)`) in an Python/Jupyter notebook. [#4197]
- Added column alignment formatting for better pprint viewing experience. [#3644]

astropy.tests

- Added new test config options, `config_dir` and `cache_dir` (these can be edited in `setup.cfg` or as extra command-line options to py.test) for setting the locations to use for the Astropy config files and download caches (see also the related `set_temp_config/cache` features added in `astropy.config`). [#3975]

astropy.time

- Add support for FITS standard time strings. [#3547]
- Allow the `format` attribute to be updated in place to change the default representation of a `Time` object. [#3673]
- Add support for shape manipulation (reshape, ravel, etc.). [#3224]
- Add argmin, argmax, argsort, min, max, ptp, sort methods. [#3681]
- Add `Time.to_datetime` method for converting `Time` objects to timezone-aware datetimes. [#4119, #4124]

astropy.units

- Added furlong to imperial units. [#3529]
- Added mil to imperial units. [#3716]
- Added stone to imperial units. [#4192]
- Added Earth Mass (`M_earth`) and Jupiter mass (`M_jup`) to units [#3907]
- Added support for functional units, in particular the logarithmic ones `Magnitude`, `Decibel`, and `Dex`. [#1894]
- Quantities now work with the unit support in matplotlib. See Plotting quantities. [#3981]
- Clarified imperial mass measurements and added pound force (lbf), kilopound (kip), and pound per square inch (psi). [#3409]

astropy.utils

- Added new `OrderedDescriptor` and `OrderedDescriptorContainer` utility classes that make it easier to implement classes with declarative APIs, wherein class-level attributes have an inherit "ordering" to them that is specified by the order in which those attributes are defined in the class declaration (by defining them using special descriptors that have `OrderedDescriptor` as a base class). See the API documentation for these classes for more details. Coordinate frames and models now use this interface. [#3679]
- The `get_pkg_data_*` functions now take an optional `package` argument which allows specifying any package to read package data filenames or content out of, as opposed to only being able to use data from

the package that the function is called from. [#4079]

- Added function `dtype_info_name` to the `data_info` module to provide the name of a `dtype` for human-readable informational purposes. [#3868]
- Added `classproperty` decorator–this is to `property` as `classmethod` is to normal instance methods. [#3982]
- `iers.open` now handles network URLs, as well as local paths. [#3850]
- The `astropy.utils.wraps` decorator now takes an optional `exclude_args` argument not shared by the standard library `wraps` decorator (as it is unique to the Astropy version's ability of copying the wrapped function's argument signature). `exclude_args` allows certain arguments on the wrapped function to be excluded from the signature of the wrapper function. This is particularly useful when wrapping an instance method as a function (to exclude the `self` argument). [#4017]
- `get_readable_fileobj` can automatically decompress LZMA ('.xz') files using the `lzma` module of Python 3.3+ or, when available, the `backports.lzma` package on earlier versions. [#3667]
- The `resolve_name` utility now accepts any number of additional positional arguments that are automatically dotted together with the first `name` argument. [#4083]
- Added `is_url_in_cache` for resolving paths to cached files via URLS and checking if files exist. [#4095]
- Added a `step` argument to the `ProgressBar.map` method to give users control over the update frequency of the progress bar. [#4191]

astropy.visualization

- Added a function / context manager `quantity_support` for enabling seamless plotting of `Quantity` instances in matplotlib. [#3981]
- Added the `hist` function, which is similar to `plt.hist` but includes several additional options for automatic determination of optimal histogram bins. This functionality was ported from the astroML library. [#3756]

astropy.wcs

- The included version of wcslib has been upgraded to 5.10. [#3992, #4239]

  The minimum required version of wcslib in the 4.x series remains 4.24.

  The minimum required version of wcslib in the 5.x series is 5.8. Building astropy against a wcslib 5.x prior to 5.8 will raise an `ImportError` when `astropy.wcs` is imported.

  The wcslib changes relevant to astropy are:

- The FITS headers returned by `astropy.wcs.WCS.to_header` and `astropy.wcs.WCS.to_header_string` now include values with more precision. This will result in numerical differences in your results if you convert `astropy.wcs.WCS` objects to FITS headers and use the results.

- `astropy.wcs.WCS` now recognises the `TPV`, `TPD`, `TPU`, `DSS`, `TNX` and `ZPX` polynomial distortions.
- Added relaxation flags to allow `PC0i_0ja`, `PV0j_0ma`, and `PS0j_0ma` (i.e. with leading zeroes on the index).
- Tidied up error reporting, particularly relating to translating status returns from lower-level functions.
- Changed output formatting of floating point values in `to_header`.
- Enhanced text representation of `WCS` objects. [#3604]
- The `astropy.tests.helper` module is now part of the public API (and has a documentation page). This module was in previous releases of astropy, but was not considered part of the public API until now. [#3890]
- There is a new function `astropy.online_help` to search the astropy documentation and display the result in a web browser. [#3642]

API changes

astropy.cosmology

- `FLRW._tfunc` and `FLRW._xfunc` are marked as deprecated. Users should use the new public interfaces `FLRW.lookback_time_integrand` and `FLRW.abs_distance_integrand` instead. [#3767]

astropy.io.ascii

- The default header line processing was made to be consistent with data line processing in that it now ignores blank lines that may have whitespace characters. Any code that explicitly specifies a `header_start` value for parsing a file with blank lines in the header containing whitespace will need to be updated. [#2654]

astropy.io.fits

- The `uint` argument to `fits.open` is now True by default; that is, arrays using the FITS unsigned integer convention will be detected, and read as unsigned integers by default. A new config option for `io.fits`, `enable_uint`, can be changed to False to revert to the original behavior of ignoring the `uint` convention unless it is explicitly requested with `uint=True`. [#3916]

- The `ImageHDU.NumCode` and `ImageHDU.ImgCode` attributes (and same for other classes derived from `_ImageBaseHDU`) are deprecated. Instead, the `astropy.io.fits` module-level constants `BITPIX2DTYPE` and `DTYPE2BITPIX` can be used. [#3916]

astropy.modeling

- Note: Comparisons of model parameters with array-like values now yields a Numpy boolean array as one would get with normal Numpy array comparison. Previously this returned a scalar True or False, with True only if the comparison was true for all elements compared, which could lead to confusing circumstances. [#3912]
- Using `model.inverse = None` to reset a model's inverse to its default is deprecated. In the future this syntax will explicitly make a model not have an inverse (even if it has a default). Instead, use `del model.inverse` to reset a model's inverse to its default (if it has a default, otherwise this just deletes any custom inverse that has been assigned to the model and is still equivalent to setting `model.inverse = None`). [#4236]
- Adds a `model.has_user_inverse` attribute which indicates whether or not a user has assigned a custom inverse to `model.inverse`. This is just for informational purposes, for example, for software that introspects model objects. [#4236]
- Renamed the parameters of `RotateNative2Celestial` and `RotateCelestial2Native` from `phi`, `theta`, `psi` to `lon`, `lat` and `lon_pole`. [#3578]
- Deprecated the `Pix2Sky_AZP.check_mu` and `Sky2Pix_AZP.check_mu` methods (these were obscure “accidentally public” methods that were probably not used by anyone). [#3910]
- Added a phase parameter to the Sine1D model. [#3807]

astropy.stats

- Renamed the `sigma_clip` `sig` keyword as `sigma`. [#3595]
- Changed the `sigma_clip` `varfunc` keyword to `stdfunc`. [#3595]
- Renamed the `sigma_clipped_stats` `mask_val` keyword to `mask_value`. [#3595]
- Changed the default `iters` keyword value to 5 in both the `sigma_clip` and `sigma_clipped_stats` functions. [#4067]

astropy.table

- `Table.as_array()` always returns a structured array with each column in the system's native byte order. The optional `keep_byteorder=True` option will keep each column’s data in its original byteorder. [#4080]

- `Table.simple_table()` now creates tables with int64 and float64 types instead of int32 and float64. [#4114]
- An empty table can now be initialized without a `names` argument as long as a valid `dtype` argument (with names embedded) is supplied. [#3977]

astropy.time

- The `astropy_time` attribute and time format has been removed from the public interface. Existing code that instantiates a new time object using `format='astropy_time'` can simply omit the `format` specification. [#3857]

astropy.units

- Single-item `Quantity` instances with record `dtype` will now have their `isscalar` property return `True`, consistent with behaviour for numpy arrays, where `np.void` records are considered scalar. [#3899]
- Three changes relating to the FITS unit format [#3993]:
- The FITS unit format will no longer parse an arbitrary number as a scale value. It must be a power of 10 of the form `10^^k`, `10^k`, `10+k`, `10-k` and `10(k)`. [#3993]
- Scales that are powers of 10 can be written out. Previously, any non-1.0 scale was rejected.
- The `*` character is accepted as a separator between the scale and the units.
- Unit formatter classes now require the `parse` and `to_string` methods are now required to be classmethods (and the formatter classes themselves are assumed to be singletons that are not instantiated). As unit formatters are mostly an internal implementation detail this is not likely to affect any users. [#4001]
- CGS E&M units are now defined separately from SI E&M units, and have distinct physical types. [#4255, #4355]

astropy.utils

- All of the `get_pkg_data_*` functions take an optional `package` argument as their second positional argument. So any code that previously passed other arguments to these functions as positional arguments might break. Use keyword argument passing instead to mitigate this. [#4079]
- `astropy.utils.iers` now uses a `QTable` internally, which means that the numerical columns are stored as `Quantity`, with full support for units. Furthermore, the `ut1_utc` method now returns a `Quantity` instead of a

float or an array (as did `pm_xy` already). [#3223]
- `astropy.utils.iers` now throws an `IERSRangeError`, a subclass of `IndexError`, rather than a raw `IndexError`. This allows more fine-grained catching of situations where a `Time` is beyond the range of the loaded IERS tables. [#4302]

astropy.wcs

- When compiled with wcslib 5.9 or later, the FITS headers returned by `astropy.wcs.WCS.to_header` and `astropy.wcs.WCS.to_header_string` now include values with more precision. This will result in numerical differences in your results if you convert `astropy.wcs.WCS` objects to FITS headers and use the results.
- If NAXIS1 or NAXIS2 is not passed with the header object to WCS.calc_footprint, a ValueError is raised. [#3557]

Bug fixes

astropy.constants

- The constants `Ry` and `u` are now properly used inside the corresponding units. The latter have changed slightly as a result. [#4229]

astropy.coordinates

- Internally, `coordinates` now consistently uses the appropriate time scales for using ERFA functions. [#4302]

astropy.io.ascii

- Fix a segfault in the fast C parser when one of the column headers is empty [#3545].
- Fix several bugs that prevented the fast readers from being used when guessing the file format. Also improved the read trace information to better understand format guessing. [#4115]
- Fix an underlying problem that resulted in an uncaught TypeError exception when reading a CDS-format file with guessing enabled. [#4120]

astropy.modeling

- `Simplex` fitter now correctly passes additional keywords arguments to the scipy solver. [#3966]
- The keyword `acc` (for accuracy) is now correctly accepted by `Simplex`. [#3966]

astropy.units

- The units `Ryd` and `u` are no longer hard-coded numbers, but depend on the appropriate values in the `constants` module. As a result, these units now imply slightly different conversions. [#4229]

Other Changes and Additions

- The `./setup.py test` command is now implemented in the `astropy.tests` module again (previously its implementation had been moved into astropy-helpers). However, that made it difficult to synchronize changes to the Astropy test runner with changes to the `./setup.py test` UI. astropy-helpers v1.1 and above will detect this implementation of the `test` command, when present, and use it instead of the old version that was included in astropy-helpers (most users will not notice any difference as a result of this change). [#4020]
- The repr for `Table` no longer displays `masked=False` since tables are not masked by default anyway. [#3869]
- The version of `PLY` that ships with astropy has been updated to 3.6.
- WCSAxes is now required for doc builds. [#4074]
- The migration guide from pre-v0.4 coordinates has been removed to avoid cluttering the `astropy.coordinates` documentation with increasingly irrelevant material. To see the migration guide, we recommend you simply look to the archived documentation for previous versions, e.g. https://docs.astropy.org/en/v1.0/coordinates/index.html#migrating-from-pre-v0-4-coordinates [#4203]
- In `astropy.coordinates`, the transformations between GCRS, CIRS, and ITRS have been adjusted to more logically reflect the order in which they actually apply. This should not affect most coordinate transformations, but may affect code that is especially sensitive to machine precision effects that change when the order in which transformations occur is changed. [#4255]
- Astropy v1.1.0 will be the last release series to officially support Python 2.6. A deprecation warning will now be issued when using Astropy in Python 2.6 (this warning can be disabled through the usual Python warning filtering mechanisms). [#3779]

1.0.13 (2017-05-29)

Bug Fixes

astropy.io.fits

- Fix use of quantize level parameter for `CompImageHDU`. [#6029]

- Prevent crash when a header contains non-ASCII (e.g. UTF-8) characters, to allow fixing the problematic cards. [#6084]

1.0.12 (2017-03-05)

Bug Fixes

astropy.convolution

- Fixed bug in `discretize_integrate_2D` in which x and y coordinates where swapped. [#5634]

astropy.coordinates

- Fixed a bug where `get_transform` could sometimes produce confusing errors because of a typo in the input validation. [#5645]

astropy.io.fits

- Guard against extremely unlikely problems in compressed images, which could lead to memory unmapping errors. [#5775]

astropy.io.votable

- Fixed a bug where stdlib `realloc()` was used instead of `PyMem_Realloc()` [#5696, #4739, #2100]

astropy.utils

- Fixed ImportError with NumPy < 1.7 and Python 3.x in `_register_patched_dtype_reduce`. [#5848]

1.0.11 (2016-12-22)

Bug Fixes

astropy.coordinates

- Initialising a SkyCoord from a list containing a single SkyCoord no longer removes the distance from the coordinate. [#5270]
- Fix errors in the implementation of the conversion to and from FK4 frames without e-terms, which will have affected coordinates not on the unit sphere (i.e., with distances). [#4293]
- Fix bug where with cds units enabled it was no longer possible to initialize an `Angle`. [#5483]

- Ensure that `search_around_sky` and `search_around_3d` return integer type index arrays for empty (non) matches. [#4877, #5083]
- Return an empty set of matches for `search_around_sky` and `search_around_3d` when one or both of the input coordinate arrays is empty. [#4875, #5083]

astropy.io.ascii

- Fix a bug with empty value at end of tab-delimited table on Windows. [#5370]
- Fix reading of big ASCII tables (more than 2Gb) with the fast reader. [#5319]
- Fix segfault with FastCsv and row with too many columns. [#5534]
- Fix problem reading an AASTex format table that does not have `\\` at the end of the last table row. [#5427]

astropy.io.fits

- Removed raising of AssertionError that could occur after closing or deleting compressed image data. [#4690, #4694, #4948]
- Fixed bug that caused an ignored exception to be displayed under certain conditions when terminating a script after using fits.getdata(). [#4977]
- Fixed usage of inplace operations that were raising an exception with recent versions of Numpy due to implicit casting. [#5250]

astropy.io.votable

- Fixed bug of `Resource.__repr__()` having undefined attributes and variables. [#5382]

astropy.modeling

- CompoundModel now correctly inherits _n_models, allowing the use of model sets [#5358]

astropy.units

- Fixed bug in Ci definition. [#5106]
- Non-ascii cds unit strings are now correctly represented using `str` also on python2. This solves bugs in parsing coordinates involving strings too. [#5355]
- Ensure `Quantity` supports `np.float_power`, which is new in numpy 1.12. [#5480]

astropy.utils

- Fixed AttributeError when calling `utils.misc.signal_number_to_name` with Python3 [#5430].

astropy.wcs

- Update the `_naxis{x}` attributes when calling `WCS.slice`. [#5411]

Other Changes and Additions

- The bundled ERFA was updated to version 1.3.0. This includes the leap second planned for 2016 Dec 31. [#5418]

1.0.10 (2016-06-09)

Bug Fixes

astropy.coordinates

- `SkyCoord` objects created before a new frame which has frame attributes is created no longer raise `AttributeError` when the new attributes are accessed [#5021]
- Fix some errors in the implementation of aberration for `get_sun`. [#4979]

astropy.io.ascii

- Fix problem reading a zero-length ECSV table with a bool type column. [#5010]

astropy.io.fits

- Fix convenience functions (`getdata`, `getheader`, `append`, `update`) to close files. [#4786]

astropy.io.votable

- The astropy.io.votable.validator.html module is updated to handle division by zero when generating validation report. [#4699]

astropy.table

- Fixed a bug where `pprint()` sometimes raises `UnicodeDecodeError` in Python 2. [#4946]
- Fix bug when doing outer join on multi-dimensional columns. [#4060]
- Fixed bug where Tables created from existing Table objects were not inheriting the `primary_key` attribute. [#4672]

astropy.tests

- Fix coverage reporting in Python 3. [#4822]

astropy.units

- Duplicates between long and short names are now removed in the `names` and `aliases` properties of units. [#5036]

astropy.utils

- The astropy.utils.xml.unescaper module now also unescapes `'%2F'` to `'/'` and `'&&'` to `'&'` in a given URL. [#4699]
- Fix two problems related to the download cache: clear_download_cache() does not work in Python 2.7 and downloading in Python 2.7 and then Python 3 can result in an exception. [#4810]

astropy.vo

- Cache option now properly caches both downloaded JSON database and XML VO tables. [#4699]
- The astropy.vo.validator.conf.conesearch_urls listing is updated to reflect external changes to some VizieR Cone Search services. [#4699]
- VOSDatabase decodes byte-string to UTF-8 instead of ASCII to avoid UnicodeDecodeError for some rare cases. Fixed a Cone Search test that is failing as a side-effect of #4699. [#4757]

Other Changes and Additions

- Updated `astropy.tests` test runner code to work with Coverage v4.0 when generating test coverage reports. [#4176]

1.0.9 (2016-03-10)

New Features

astropy.nddata

- `NDArithmeticMixin` check for matching WCS now works with `astropy.wcs.WCS` objects [#4499, #4503]

Bug Fixes

astropy.convolution

- Correct a bug in which `psf_pad` and `fft_pad` would be ignored [#4366]

astropy.io.ascii

- Fixed addition of new line characters after last row of data in ascii.latex.AASTex. [#4561]
- Fixed reading of Latex tables where the `\tabular` tag is in the first line. [#4595]
- Fix use of plain format strings with the fast writer. [#4517]
- Fix bug writing space-delimited file when table has empty fields. [#4417]

astropy.io.fits

- Fixed possible segfault during error handling in FITS tile compression. [#4489]
- Fixed crash on pickling of binary table columns with the 'X', 'P', or 'Q' format. [#4514]
- Fixed memory / reference leak that could occur when copying a `FITS_rec` object (the `.data` for table HDUs). [#520]
- Fixed a memory / reference leak in `FITS_rec` that occurred in a wide range of cases, especially after writing FITS tables to a file, but in other cases as well. [#4539]

astropy.modeling

- Fixed display of compound model expressions and components when printing compound model instances. [#4414, #4482]

astropy.stats

- the input for median_absolute_deviation will not be cast to plain numpy arrays when given subclasses of numpy arrays (like Quantity, numpy.ma.MaskedArray, etc.) [#4658]
- Fixed incorrect results when using median_absolute_deviation with masked arrays. [#4658]

astropy.utils

- The `zest.releaser` hooks included in Astropy are now injected locally to Astropy, rather than being global. [#4650]

astropy.visualization

- Fixed `fits2bitmap` script to allow ext flag to contain extension names or numbers. [#4468]
- Fixed `fits2bitmap` default output filename generation for compressed FITS files. [#4468]

1.0.8 (2016-01-08)

Bug Fixes

astropy.io.fits

- Fixed an bug where updates to string columns in FITS tables were not saved on Python 3. [#4452]

astropy.units

- In-place peak-to-peak calculations now work on `Quantity`. [#4442]

astropy.utils

- Fixed `find_api_page` to work correctly on python 3.x [#4378, #4379]

1.0.7 (2015-12-04)

Bug Fixes

astropy.coordinates

- Pickling of `EarthLocation` instances now also works on Python 2. [#4304]

astropy.io.ascii

- Fix fast writer so bytestring column output is not prefixed by ‘b’ in Python 3. [#4350]

astropy.io.fits

- Fixed a regression that could cause writes of large FITS files to be truncated. [#4307]
- Astropy v1.0.6 included a fix (#4228) for an obscure case where the TDIM of a table column is smaller than the repeat count of its data format. This updates that fix in such a way that it works with Numpy 1.10 as well. [#4266]

astropy.table

- Fix a bug when pickling a Table with mixin columns (e.g. Time). [#4098]

astropy.time

- Fix incorrect `value` attribute for epoch formats like “unix” when `scale` is different from the class `epoch_scale`. [#4312]

astropy.utils

- Fixed an issue where if ipython is installed but ipykernel is not installed then importing astropy from the ipython console gave an IPython.kernel deprecation warning. [#4279]
- Fixed crash that could occur in `ProgressBar` when `astropy` is imported in an IPython startup script. [#4274]

Other Changes and Additions

- Updated bundled astropy-helpers to v1.0.6. [#4372]

1.0.6 (2015-10-22)

Bug Fixes

astropy.analytic_functions

- Fixed blackbody analytic functions to properly support arrays of temperatures. [#4251]

astropy.coordinates

- Fixed errors in transformations for objects within a few AU of the Earth. Included substantive changes to transformation machinery that may change distances at levels ~machine precision for other objects. [#4254]

astropy.io.fits

- `fitsdiff` and related functions now do a better job reporting differences between values that are different types but have the same representation (ex: the string ‘0’ versus the number 0). [#4122]
- Miscellaneous fixes for supporting Numpy 1.10. [#4228]
- Fixed an issue where writing a column of unicode strings to a FITS table resulted in a quadrupling of size of the column (i.e. the format of the FITS column was 4 characters for every one in the original strings). [#4228]
- Added support for an obscure case (but nonetheless allowed by the FITS standard) where a column has some TDIMn keyword, but a repeat count in the TFORMn column greater than the number of elements implied by the TDIMn. For example TFORMn = 100I, but TDIMn = ‘(5,5)’. In this case the TDIMn implies 5x5 arrays in the column, but the TFORMn implies a 100 element 1-D array in the column. In this case the TDIM takes precedence, and the remaining bytes in the column are ignored. [#4228]

astropy.io.votable

- Fixed crash with Python compiler optimization level = 2. [#4231]

astropy.vo

- Fixed `check_conesearch_sites` with `parallel=True` on Python >= 3.3 and on Windows (it was broken in both those cases for separate reasons). [#2970]

Other Changes and Additions

- All tests now pass against Numpy v1.10.x. This implies nominal support for Numpy 1.10.x moving forward (but there may still be unknown issues). For example, there is already a known performance issue with tables containing large multi-dimensional columns–for example, tables that contain entire images in one or more of their columns. This is a known upstream issue in Numpy. [#4259]

1.0.5 (2015-10-05)

Bug Fixes

astropy.constants

- Rename units -> unit and error -> uncertainty in the `repr` and `str` of constants to match attribute names. [#4147]

astropy.coordinates

- Fix string representation of `SkyCoord` objects transformed into the `AltAz` frame [#4055, #4057]
- Fix the `search_around_sky` function to allow `storekdtree` to be `False` as was intended. [#4082, #4212]

astropy.io.fits

- Fix bug when extending one header (without comments) with another (with comments). [#3967]
- Somewhat improved resource usage for FITS data–previously a new `mmap` was opened for each HDU of a FITS file accessed through an `HDUList`. Each `mmap` used up a single file descriptor, causing problems with system resource limits for some users. Now only a single `mmap` is opened, and shared for the data of all HDUs. Note: The problem still persists with using the “convenience” functions. For example using `fits.getdata` will create one `mmap` per HDU read this way (as opposed to opening the file with

`fits.open` and accessing the HDUs through the `HDUList` object). [#4097]
- Fix bug where reading a file without a newline failed with an unrelated / unhelpful exception. [#4160]

astropy.modeling

- Cleaned up `repr` of models that have no parameters. [#4076]

astropy.nddata

- Initializing `NDDataArray` from another instance now sets `flags` as expected and no longer fails when `uncertainty` is set [#4129]. Initializing an `NDData` subclass from a parent instance (eg. `NDDataArray` from `NDData`) now sets the attributes other than `data` as it should [#4130, #4137].

astropy.table

- Fix an issue with setting fill value when column dtype is changed. [#4088]
- Fix bug when unpickling a bare Column where the _parent_table attribute was not set. This impacted the Column representation. [#4099]
- Fix issue with the web browser opening with an empty page, and ensure that the url is correctly formatted for Windows. [#4132]
- Fix NameError in table stack exception message. [#4213]

astropy.utils

- `resolve_name` no longer causes `sys.modules` to be cluttered with additional copies of modules under a package imported like `resolve_name('numpy')`. [#4084]
- `console` was updated to support IPython 4.x and Jupyter 1.x. This should suppress a ShimWarning that was appearing at import of astropy with IPython 4.0 or later. [#4078]
- Temporary downloaded files created by `get_readable_fileobj` when passed a URL are now deleted immediately after the file is closed. [#4198]

astropy.visualization

- The color for axes labels was set to white. Since white labels on white background are hard to read, the label color has been changed to black. [#4143]
- `ImageNormalize` now automatically determines `vmin`/`vmax` (via the `autoscale_None` method) when they have not been set explicitly. [#4117]

astropy.vo

- Cone Search validation no longer crashes when the provider gives an incomplete test query. It also ensures search radius for a test query is not too large to avoid timeout. [#4158, #4159]

Other Changes and Additions

- Astropy now supports Python 3.5. [#4027]
- Updated bundled version of astropy-helpers to 1.0.5. [#4215]
- Updated tests to support py.test 2.7, and upgraded the bundled copy of py.test to v2.7.3. [#4027]

1.0.4 (2015-08-11)

New Features

astropy.convolution

- Modified Cython functions to release the GIL. This enables convolution to be parallelized effectively and gives large speedups when used with multithreaded task schedulers such as Dask. [#3949]

API Changes

astropy.coordinates

- Some transformations for an input coordinate that's a scalar now correctly return a scalar. This was always the intended behavior, but it may break code that has been written to work-around this bug, so it may be viewed as an unplanned API change [#3920, #4039]

astropy.visualization

- The `astropy_mpl_style` no longer sets `interactive` to `True`, but instead leaves it at the user preference. This makes using the style compatible with building docs with Sphinx, and other non-interactive contexts. [#4030]

Bug Fixes

astropy.coordinates

- Fix bug where coordinate representation setting gets reset to default value when coordinate array is indexed or sliced. [#3824]

- Fixed confusing warning message shown when using dates outside current IERS data. [#3844]
- `get_sun` now yields a scalar when the input time is a scalar (this was a regression in v1.0.3 from v1.0.2) [#3998, #4039]
- Fixed bug where some scalar coordinates were incorrectly being changed to length-1 array coordinates after transforming through certain frames. [#3920, #4039]
- Fixed bug causing the `separation` methods of `SkyCoord` and frame classes to fail due to infinite recursion [#4033, #4039]
- Made it so that passing in a list of `SkyCoord` objects that are in UnitSphericalRepresentation to the `SkyCoord` constructor appropriately yields a new object in UnitSphericalRepresentation [#3938, #4039]

astropy.cosmology

- Fixed wCDM to not ignore the Ob0 parameter on initialization. [#3934]

astropy.io.fits

- Fixed crash when updating data in a random groups HDU opened in update mode. [#3730]
- Fixed incorrect checksum / datasum being written when re-writing a scaled HDU (i.e. non-trivial BSCALE and/or BZERO) with `do_not_scale_image_data=False`. [#3883]
- Fixed stray deprecation warning in `BinTableHDU.copy()`. [#3798]
- Better handling of the `BLANK` keyword when auto-scaling scaled image data. The `BLANK` keyword is now removed from the header after auto-scaling is applied, and it is restored properly (with floating point NaNs replaced by the filler value) when updating a file opened with the `scale_back=True` argument. Invalid usage of the `BLANK` keyword is also better warned about during validation. [#3865]
- Reading memmaped scaled images won’t fail when `do_not_scale_image_data=True` (that is, since we’re just reading the raw / physical data there is no reason mmap can’t be used). [#3766]
- Fixed a reference cycle that could sometimes cause FITS table-related objects (`BinTableHDU`, `ColDefs`, etc.) to hang around in memory longer than expected. [#4012]

astropy.modeling

- Improved support for pickling of compound models, including both compound model instances, and new compound model classes. [#3867]
- Added missing default values for `Ellipse2D` parameters. [#3903]

astropy.time

- Fixed iteration of scalar `Time` objects so that `iter()` correctly raises a `TypeError` on them (while still allowing `Time` arrays to be iterated). [#4048]

astropy.units

- Added frequency-equivalency check when declaring doppler equivalencies [#3728]
- Define `floor_divide` (`//`) for `Quantity` to be consistent `divmod`, such that it only works where the quotient is dimensionless. This guarantees that `(q1 // q2) * q2 + (q1 % q2) == q1`. [#3817]
- Fixed the documentation of supported units to correctly report support for SI prefixes. Previously the table of supported units incorrectly showed several derived unit as not supporting prefixes, when in fact they do. [#3835]
- Fix a crash when calling `astropy.units.cds.enable()`. This will now "set" rather than "add" units to the active set to avoid the namespace clash with the default units. [#3873]
- Ensure in-place operations on `float32` quantities work. [#4007]

astropy.utils

- The `deprecated` decorator did not correctly wrap classes that have a custom metaclass–the metaclass could be dropped from the deprecated version of the class. [#3997]
- The `wraps` decorator would copy the wrapped function's name to the wrapper function even when `'__name__'` is excluded from the `assigned` argument. [#4016]

Misc

- `fitscheck` no longer causes scaled image data to be rescaled when adding checksums to existing files. [#3884]
- Fixed an issue where running `import astropy` from within the source tree did not automatically build the extension modules if the source is from a source distribution (as opposed to a git repository). [#3932]
- Fixed multiple instances of a bug that prevented Astropy from being used when compiled with the `python -OO` flag, due to it causing all docstrings to be stripped out. [#3923]
- Removed source code template files that were being installed accidentally alongside installed Python modules. [#4014]
- Fixed a bug in the exception logging that caused a crash in the exception handler itself on Python 3 when exceptions do not include a message.

[#4056]

1.0.3 (2015-06-05)

New Features

astropy.table

- Greatly improved the speed of printing a large table to the screen when only a few rows are being displayed. [#3796]

astropy.time

- Add support for the 2015-Jun-30 leap second. [#3794]

API Changes

astropy.io.ascii

- Note that HTML formatted tables will not always be found with guess mode unless it passes certain heuristics that strongly suggest the presence of HTML in the input. Code that expects to read tables from HTML should specify `format='html'` explicitly. See bug fixes below for more details. [#3693]

Bug Fixes

astropy.convolution

- Fix issue with repeated normalizations of `Kernels`. [#3747]

astropy.coordinates

- Fixed `get_sun` to yield frames with the `obstime` set to what's passed into the function (previously it incorrectly always had J2000). [#3750]
- Fixed `get_sun` to account for aberration of light. [#3750]
- Fixed error in the GCRS->ICRS transformation that gave incorrect distances. [#3750]

astropy.io.ascii

- Remove HTML from the list of automatically-guessed formats when reading if the file does not appear to be HTML. This was necessary to avoid a commonly-encountered segmentation fault occurring in the libxml parser on MacOSX. [#3693]

astropy.io.fits

- Fixes to support the upcoming Numpy 1.10. [#3419]

astropy.modeling

- Polynomials are now scaled when used in a compound model. [#3702]
- Fixed the `Ellipse2D` model to be consistent with `Disk2D` in how pixels are included. [#3736]
- Fixed crash when evaluating a model that accepts no inputs. [#3772]

astropy.testing

- The Astropy py.test plugins that disable unintentional internet access in tests were also blocking use of local UNIX sockets in tests, which prevented testing some multiprocessing code–fixed. [#3713]

astropy.units

- Supported full SI prefixes for the barn unit (“picobarn”, “femtobarn”, etc.) [#3753]
- Fix loss of precision when multiplying non-whole-numbered powers of units together. For example, before this change, `(u.m ** 1.5) ** Fraction(4, 5)` resulted in an inaccurate floating-point power of `1.2000000000000002`. After this change, the exact rational number of `Fraction(6, 5)` is maintained. [#3790]
- Fixed printing of object ndarrays containing multiple Quantity objects with differing / incompatible units. Note: Unit conversion errors now cause a `UnitConversionError` exception to be raised. However, this is a subclass of the `UnitsError` exception used previously, so existing code that catches `UnitsError` should still work. [#3778]

Other Changes and Additions

- Added a new `astropy.__bibtex__` attribute which gives a citation for Astropy in bibtex format. [#3697]
- The bundled version of ERFA was updated to v1.2.0 to address leapsecond updates. [#3802]

0.4.6 (2015-05-29)

Bug Fixes

astropy.time

- Fixed ERFA code to handle the 2015-Jun-30 leap second. [#3795]

1.0.2 (2015-04-16)

New Features

astropy.modeling

- Added support for polynomials with degree 0 or degree greater than 15. [#3574, 3589]

Bug Fixes

astropy.config

- The pre-astropy-0.4 configuration API has been fixed. It was inadvertently broken in 1.0.1. [#3627]

astropy.io.fits

- Fixed a severe memory leak that occurred when reading tile compressed images. [#3680]
- Fixed bug where column data could be unintentionally byte-swapped when copying data from an existing FITS file to a new FITS table with a TDIMn keyword for that column. [#3561]
- The `ColDefs.change_attrib`, `ColDefs.change_name`, and `ColDefs.change_unit` methods now work as advertised. It is also possible (and preferable) to update attributes directly on `Column` objects (for example setting `column.name`), and the change will be accurately reflected in any associated table data and its FITS header. [#3283, #1539, #2618]
- Fixes an issue with the `FITS_rec` interface to FITS table data, where a `FITS_rec` created by copying an existing FITS table but adding new rows could not be sliced or masked correctly. [#3641]
- Fixed handling of BINTABLE with TDIMn of size 1. [#3580]

astropy.io.votable

- Loading a `TABLE` element without any `DATA` now correctly creates a 0-row array. [#3636]

astropy.modeling

- Added workaround to support inverses on compound models when one of the sub-models is itself a compound model with a manually-assigned custom

inverse. [#3542]
- Fixed instantiation of polynomial models with constraints for parameters (constraints could still be assigned after instantiation, but not during). [#3606]
- Fixed fitting of 2D polynomial models with the `LeVMarLSQFitter`. [#3606]

astropy.table

- Ensure `QTable` can be pickled [#3590]
- Some corner cases when instantiating an `astropy.table.Table` with a Numpy array are handled [#3637]. Notably:
- a zero-length array is the same as passing `None`
- a scalar raises a `ValueError`
- a one-dimensional array is treated as a single row of a table.
- Ensure a `Column` without units is treated as an `array`, not as an dimensionless `Quantity`. [#3648]

astropy.units

- Ensure equivalencies that do more than just scale a `Quantity` are properly handled also in `ufunc` evaluations. [#2496, #3586]
- The LaTeX representation of the Angstrom unit has changed from `\overset{\circ}{A}` to `\mathring{A}`, which should have better support across regular LaTeX, MathJax and matplotlib (as of version 1.5) [#3617]

astropy.vo

- Using HTTPS/SSL for communication between SAMP hubs now works correctly on all supported versions of Python [#3613]

astropy.wcs

- When no `relax` argument is passed to `WCS.to_header()` and the result omits non-standard WCS keywords, a warning is emitted. [#3652]

Other Changes and Additions

astropy.vo

- The number of retries for connections in `astropy.vo.samp` can now be configured by a `n_retries` configuration option. [#3612]
- Testing
- Running `astropy.test()` from within the IPython prompt has been provisionally re-enabled. [#3184]

1.0.1 (2015-03-06)

Bug Fixes

astropy.constants

- Ensure constants can be turned into `Quantity` safely. [#3537, #3538]

astropy.io.ascii

- Fix a segfault in the fast C parser when one of the column headers is empty [#3545].
- Fixed support for reading inf and nan values with the fast reader in Windows. Also fixed in the case of using `use_fast_converter=True` with the fast reader. [#3525]
- Fixed use of mmap in the fast reader on Windows. [#3525]
- Fixed issue where commented header would treat comments defining the table (i.e. column headers) as purely information comments, leading to problems when trying to round-trip the table. [#3562]

astropy.modeling

- Fixed propagation of parameter constraints ('fixed', 'bounds', 'tied') between compound models and their components. There is may still be some difficulty defining 'tied' constraints properly for use with compound models, however. [#3481]

astropy.nddata

- Restore several properties to the compatibility class `NDDataArray` that were inadvertently omitted [#3466].

astropy.time

- Time objects now always evaluate to `True`, except when empty. [#3530]

Miscellaneous

- The ERFA wrappers are now written directly in the Python/C API rather than using Cython, for greater performance. [#3521]
- Improve import time of astropy [#3488].

Other Changes and Additions

- Updated bundled astropy-helpers version to v1.0.1 to address installation issues with some packages that depend on Astropy. [#3541]

1.0 (2015-02-18)

General

- Astropy now requires Numpy 1.6.0 or later.

New Features

astropy.analytic_functions

- The `astropy.analytic_functions` was added to contain analytic functions useful for astronomy [#3077].

astropy.coordinates

- `astropy.coordinates` now has a full stack of frames allowing transformations from ICRS or other celestial systems down to Alt/Az coordinates. [#3217]
- `astropy.coordinates` now has a `get_sun` function that gives the coordinates of the Sun at a specified time. [#3217]
- `SkyCoord` now has `to_pixel` and `from_pixel` methods that convert between celestial coordinates as `SkyCoord` objects and pixel coordinates given an `astropy.wcs.WCS` object. [#3002]
- `SkyCoord` now has `search_around_sky` and `search_around_3d` convenience methods that allow searching for all coordinates within a certain distance of another `SkyCoord`. [#2953]
- `SkyCoord` can now accept a frame instance for the `frame=` keyword argument. [#3063]
- `SkyCoord` now has a `guess_from_table` method that can be used to quickly create `SkyCoord` objects from an `astropy.table.Table` object. [#2951]
- `astropy.coordinates` now has a `Galactocentric` frame, a coordinate frame centered on a (user specified) center of the Milky Way. [#2761, #3286]
- `SkyCoord` now accepts more formats of the coordinate string when the representation has `ra` and `dec` attributes. [#2920]
- `SkyCoord` can now accept lists of `SkyCoord` objects, frame objects, or representation objects and will combine them into a single object. [#3285]
- Frames and `SkyCoord` instances now have a method `is_equivalent_frame` that can be used to check that two frames are equivalent (ignoring the data). [#3330]
- The `__repr__` of coordinate objects now shows scalar coordinates in the

same format as vector coordinates. [#3350, 3448]

astropy.cosmology

- Added `lookback_distance`, which is `c * lookback_time`. [#3145]
- Add baryonic matter density and dark matter only density parameters to cosmology objects [#2757].
- Add a `clone` method to cosmology objects to allow copies of cosmological objects to be created with the specified variables modified [#2592].
- Increase default numerical precision of `z_at_value` following the accurate by default, fast by explicit request model [#3074].
- Cosmology functions that take a single (redshift) input now broadcast like numpy ufuncs. So, passing an arbitrarily shaped array of inputs will produce an output of the same shape. [#3178, #3194]

astropy.io.ascii

- Simplify the way new Reader classes are defined, allowing custom behavior entirely by overriding inherited class attributes instead of setting instance attributes in the Reader `__init__` method. [#2812]
- There is now a faster C/Cython engine available for reading and writing simple ASCII formats like CSV. Both are enabled by default, and fast reading will fall back on an ordinary reader in case of a parsing failure. Their behavior can be altered with the parameter `fast_reader` in `read` and `fast_writer` in `write`. [#2716]
- Make Latex/AASTex tables use unit attribute of Column for output. [#3064]
- Store comment lines encountered during reading in metadata of the output table via `meta['comment_lines']`. [#3222]
- Write comment lines in Table metadata during output for all basic formats, IPAC, and fast writers. This functionality can be disabled with `comment=False`. [#3255]
- Add reader / writer for the Enhanced CSV format which stores table and column meta data, in particular data type and unit. [#2319]

astropy.io.fits

- The `fitsdiff` script ignores some things by default when comparing fits files (e.g. empty header lines). This adds a `--exact` option where nothing is ignored. [#2782, #3110]
- The `fitsheader` script now takes a `--keyword` option to extract a specific keyword from the header of a FITS file, and a `--table` option to export headers into any of the data formats supported by `astropy.table`. [#2555, #2588]
- `Section` now supports all advanced indexing features `ndarray` does

(slices with any steps, integer arrays, boolean arrays, None, Ellipsis). It also properly returns scalars when this is appropriate. [#3148]

astropy.io.votable

- `astropy.io.votable.parse` now takes a `datatype_mapping` keyword argument to map invalid datatype names to valid ones in order to support non-compliant files. [#2675]

astropy.modeling

- Added the capability of creating new "compound" models by combining existing models using arithmetic operators. See the "What's New in 1.0" page in the Astropy documentation for more details. [#3231]
- A new `custom_model` decorator/factory function has been added for converting normal functions to `Model` classes that can work within the Astropy modeling framework. This replaces the old `custom_model_1d` function which is now deprecated. The new function works the same as the old one but is less limited in the types of models it can be used to created. [#1763]
- The `Model` and `Fitter` classes have `.registry` attributes which provide sets of all loaded `Model` and `Fitter` classes (this is useful for building UIs for models and fitting). [#2725]
- A dict-like `meta` member was added to `Model`. it is to be used to store any optional information which is relevant to a project and is not in the standard `Model` class. [#2189]
- Added `Ellipse2D` model. [#3124]

astropy.nddata

- New array-related utility functions in `astropy.nddata.utils` for adding and removing arrays from other arrays with different sizes/shapes. [#3201]
- New metaclass `NDDataBase` for enforcing the nddata interface in subclasses without restricting implementation of the data storage. [#2905]
- New mixin classes `NDSlicingMixin` for slicing, `NDArithmeticMixin` for arithmetic operations, and `NDIOMixin` for input/ouput in NDData. [#2905]
- Added a decorator `support_nddata` that can be used to write functions that can either take separate arguments or NDData objects. [#2855]

astropy.stats

- Added `mad_std()` function. [#3208]
- Added `gaussian_fwhm_to_sigma` and `gaussian_sigma_to_fwhm`

constants. [#3208]
- New function `sigma_clipped_stats` which can be used to quickly get common statistics for an array, using sigma clipping at the same time. [#3201]

astropy.table

- Changed the internal implementation of the `Table` class changed so that it no longer uses numpy structured arrays as the core table data container. [#2790, #3179]
- Tables can now be written to an html file that includes interactive browsing capabilities. To write out to this format, use `Table.write('filename.html', format='jsviewer')`. [#2875]
- A `quantity` property and `to` method were added to `Table` columns that allow the column values to be easily converted to `astropy.units.Quantity` objects. [#2950]
- Add `unique` convenience method to table. [#3185]

astropy.tests

- Added a new Quantity-aware `assert_quantity_allclose`. [#3273]

astropy.time

- `Time` can now handle arbitrary array dimensions, with operations following standard numpy broadcasting rules. [#3138]

astropy.units

- Support for VOUnit has been updated to be compliant with version 1.0 of the standard. [#2901]
- Added an `insert` method to insert values into a `Quantity` object. This is similar to the `numpy.insert` function. [#3049]
- When viewed in IPython, `Quantity` objects with array values now render using LaTeX and scientific notation. [#2271]
- Added `units.quantity_input` decorator to validate quantity inputs to a function for unit compatibility. [#3072]
- Added `units.astronomical_unit` as a long form for `units.au`. [#3303]

astropy.utils

- Added a new decorator `astropy.utils.wraps` which acts as a replacement for the standard library's `functools.wraps`, the only difference being that the decorated function also preserves the wrapped function’s call signature. [#2849]

- `astropy.utils.compat.numpy` has been revised such that it can include patched versions of routines from newer `numpy` versions. The first addition is a version of `broadcast_arrays` that can be used with `Quantity` and other `ndarray` subclasses (using the `subok=True` flag). [#2327]
- Added `astropy.utils.resolve_name` which returns a member of a module or class given the fully qualified dotted name of that object as a string. [#3389]
- Added `astropy.utils.minversion` which can be used to check minimum version requirements of Python modules (to test for specific features and/ or bugs and the like). [#3389]

astropy.visualization

- Created `astropy.visualization` module and added functionality relating to image normalization (i.e. stretching and scaling) as well as a new script `fits2bitmap` that can produce a bitmap image from a FITS file. [#3201]
- Added dictionary `astropy.visualization.mpl_style.astropy_mpl_style` which can be used to set a uniform plotstyle specifically for tutorials that is improved compared to matplotlib defaults. [#2719, #2787, #3200]

astropy.wcs

- `wcslib` has been upgraded to version 4.25. This brings a single new feature:
- `equinox` and `radesys` will now be given default values conforming with the WCS specification if `EQUINOXa` and `RADESYSa`, respectively, are not present in the header.
- The minimum required version of `wcslib` is now 4.24. [#2503]
- Added a new function `wcs_to_celestial_frame` that can be used to find the astropy.coordinates celestial frame corresponding to a particular WCS. [#2730]
- `astropy.wcs.WCS.compare` now supports a `tolerance` keyword argument to allow for approximate comparison of floating-point values. [#2503]
- added `pixel_scale_matrix`, `celestial`, `is_celestial`, and `has_celestial` convenience attributes. Added `proj_plane_pixel_scales`, `proj_plane_pixel_area`, and `non_celestial_pixel_scales` utility functions for retrieving WCS pixel scale and area information [#2832, #3304]

- Added two functions `pixel_to_skycoord` and `skycoord_to_pixel` that make it easy to convert between SkyCoord objects and pixel coordinates. [#2885]
- `all_world2pix` now uses a much more sophisticated and complete algorithm to iteratively compute the inverse WCS transform. [#2816]
- Add ability to use `WCS` object to define projections in Matplotlib, using the `WCSAxes` package. [#3183]
- Added `is_proj_plane_distorted` for testing if pixels are distorted. [#3329]

Misc

- `astropy._erfa` was added as a new subpackage wrapping the functionality of the ERFA library in python. This is primarily of use for other astropy subpackages, but the API may be made more public in the future. [#2992]

API Changes

astropy.coordinates

- Subclasses of `BaseCoordinateFrame` which define a custom `repr` should be aware of the format expected in `SkyCoord.__repr__()`, which changed in this release. [#2704, #2882]
- The `CartesianPoints` class (deprecated in v0.4) has now been removed. [#2990]
- The previous `astropy.coordinates.builtin_frames` module is now a subpackage. Everything that was in the `astropy.coordinates.builtin_frames` module is still accessible from the new package, but the classes are now in separate modules. This should have no direct impact at the user level. [#3120]
- Support for passing a frame as a positional argument in the `SkyCoord` class has now been deprecated, except in the case where a frame with data is passed as the sole positional argument. [#3152]
- Improved `__repr__` of coordinate objects representing a single coordinate point for the sake of easier copy/pasting. [#3350]

astropy.cosmology

- The functional interface to the cosmological routines as well as `set_current` and `get_current` (deprecated in v0.4) have now been removed. [#2990]

astropy.io.ascii

- Added a new argument to `htmldict` in the HTML reader named `parser`, which allows the user to specify which parser BeautifulSoup should use as a backend. [#2815]
- Add `FixedWidthTwoLine` reader to guessing. This will allows to read tables that a copied from screen output like `print my_table` to be read automatically. Discussed in #3025 and #3099 [#3109]

astropy.io.fits

- A new optional argument `cache` has been added to `astropy.io.fits.open()`. When opening a FITS file from a URL, `cache` is a boolean value specifying whether or not to save the file locally in Astropy's download cache (`True` by default). [#3041]

astropy.modeling

- Model classes should now specify `inputs` and `outputs` class attributes instead of the old `n_inputs` and `n_outputs`. These should be tuples providing human-readable *labels* for all inputs and outputs of the model. The length of the tuple indicates the numbers of inputs and outputs. See "What's New in Astropy 1.0" for more details. [#2835]
- It is no longer necessary to include `__init__` or `__call__` definitions in `Model` subclasses if all they do is wrap the super-method in order to provide a nice call signature to the docs. The `inputs` class attribute is now used to generate a nice call signature, so these methods should only be overridden by `Model` subclasses in order to provide new functionality. [#2835]
- Most models included in Astropy now have sensible default values for most or all of their parameters. Call `help(ModelClass)` on any model to check what those defaults are. Most of them time they should be overridden, but some of them are useful (for example spatial offsets are always set at the origin by default). Another rule of thumb is that, where possible, default parameters are set so that the model is a no-op, or close to it, by default. [#2932]
- The `Model.inverse` method has been changed to a *property*, so that now accessing `model.inverse` on a model returns a new model that implements that model's inverse, and *calling* ``model.inverse(...)` `` on some independent variable computes the value of the inverse (similar to what the old `Model.invert()` method was meant to do). [#3024]
- The `Model.invert()` method has been removed entirely (it was never implemented and there should not be any existing code that relies on it). [#3024]

- `custom_model_1d` is deprecated in favor of the new `custom_model` (see “New Features” above). [#1763]
- The `Model.param_dim` property (deprecated in v0.4) has now been removed. [#2990]
- The `Beta1D` and `Beta2D` models have been renamed to `Moffat1D` and `Moffat2D`. [#3029]

astropy.nddata

- `flags`, `shape`, `size`, `dtype` and `ndim` properties removed from `astropy.nddata.NDData`. [#2905]
- Arithmetic operations, uncertainty propagation, slicing and automatic conversion to a numpy array removed from `astropy.nddata.NDData`. The class `astropy.nddata.NDDataArray` is functionally equivalent to the old `NDData`. [#2905]

astropy.table

- The `Column.units` property (deprecated in v0.3) has now been removed. [#2990]
- The `Row.data` and `Table._data` attributes have been deprecated related to the change in Table implementation. They are replaced by `Row.as_void()` and `Table.as_array()` methods, respectively. [#2790]
- The `Table.create_mask` method has been removed. This undocumented method was a development orphan and would cause corruption of the table if called. [#2790]
- The return type for integer item access to a Column (e.g. col[12] or t[‘a’][12]) is now always a numpy scalar, numpy `ndarray`, or numpy `MaskedArray`. Previously if the column was multidimensional then a Column object would be returned. [#3095]
- The representation of Table and Column objects has been changed to be formatted similar to the print output. [#3239]

astropy.time

- The `Time.val` and `Time.vals` properties (deprecated in v0.3) and the `Time.lon`, and `Time.lat` properties (deprecated in v0.4) have now been removed. [#2990]
- Add `decimalyear` format that represents time as a decimal year. [#3265]

astropy.units

- Support for VOUnit has been updated to be compliant with version 1.0 of the standard. This means that some VOUnit strings that were rejected before are now acceptable. [#2901] Notably:

- SI prefixes are supported on most units
- Binary prefixes are supported on “bits” and “bytes”
- Custom units can be defined “inline” by placing them between single quotes.
- `Unit.get_converter` has been deprecated. It is not strictly necessary for end users, and it was confusing due to lack of support for `Quantity` objects. [#3456]

astropy.utils

- Some members of `astropy.utils.misc` were moved into new submodules. Specifically:
- `deprecated`, `deprecated_attribute`, and `lazyproperty` -> `astropy.utils.decorators`
- `find_current_module`, `find_mod_objs` -> `astropy.utils.introspection`

  All of these functions can be imported directly from `astropy.utils` which should be preferred over referencing individual submodules of `astropy.utils`. [#2857]
- The ProgressBar.iterate class method (deprecated in v0.3) has now been removed. [#2990]
- Updated `astropy/utils/console.py` ProgressBar() module to display output to IPython notebook with the addition of an `interactive` kwarg. [#2658, #2789]

astropy.wcs

- The `WCS.calcFootprint` method (deprecated in v0.4) has now been removed. [#2990]
- An invalid unit in a `CUNITn` keyword now displays a warning and returns a `UnrecognizedUnit` instance rather than raising an exception [#3190]

Bug Fixes

astropy.convolution

- `astropy.convolution.discretize_model` now handles arbitrary callables correctly [#2274].

astropy.coordinates

- `Angle.to_string` now outputs unicode arrays instead of object arrays. [#2981]
- `SkyCoord.to_string` no longer gives an error when used with an array coordinate with more than one dimension. [#3340]
- Fixed support for subclasses of `UnitSphericalRepresentation` and `SphericalRepresentation` [#3354, #3366]
- Fixed latex display of array angles in IPython notebook. [#3480]

astropy.io.ascii

- In the `CommentedHeader` the `data_start` parameter now defaults to `0`, which is the first uncommented line. Discussed in #2692. [#3054]
- Position lines in `FixedWidthTwoLine` reader could consist of many characters. Now, only one character in addition to the delimiter is allowed. This bug was discovered as part of [#3109]
- The IPAC table writer now consistently uses the `fill_values` keyword to specify the output null values. Previously the behavior was inconsistent or incorrect. [#3259]
- The IPAC table reader now correctly interprets abbreviated column types. [#3279]
- Tables that look almost, but not quite like DAOPhot tables could cause guessing to fail. [#3342]

astropy.io.fits

- Fixed the problem in `fits.open` of some filenames with colon (`:`) in the name being recognized as URLs instead of file names. [#3122]
- Setting `memmap=True` in `fits.open` and related functions now raises a ValueError if opening a file in memory-mapped mode is impossible. [#2298]
- CONTINUE cards no longer end the value of the final card in the series with an ampersand, per the specification of the CONTINUE card convention. [#3282]
- Fixed a crash that occurred when reading an ASCII table containing zero-precision floating point fields. [#3422]
- When a float field for an ASCII table has zero-precision a decimal point (with no digits following it) is still written to the field as long as there is space for it, as recommended by the FITS standard. This makes it less ambiguous that these columns should be interpreted as floats. [#3422]

astropy.logger

- Fix a bug that occurred when displaying warnings that produced an error message `dictionary changed size during iteration`. [#3353]

astropy.modeling

- Fixed a bug in `SLSQPLSQFitter` where the `maxiter` argument was not passed correctly to the optimizer. [#3339]

astropy.table

- Fix a problem where `table.hstack` fails to stack multiple references to the same table, e.g. `table.hstack([t, t])`. [#2995]
- Fixed a problem where `table.vstack` and `table.hstack` failed to stack a single table, e.g. `table.vstack([t])`. [#3313]
- Fix a problem when doing nested iterators on a single table. [#3358]
- Fix an error when an empty list, tuple, or ndarray is used for item access within a table. This now returns the table with no rows. [#3442]

astropy.time

- When creating a Time object from a datetime object the time zone info is now correctly used. [#3160]
- For Time objects, it is now checked that numerical input is finite. [#3396]

astropy.units

- Added a `latex_inline` unit format that returns the units in LaTeX math notation with negative exponents instead of fractions [#2622].
- When using a unit that is deprecated in a given unit format, non-deprecated alternatives will be suggested. [#2806] For example:

```
>>> import astropy.units as u
>>> u.Unit('Angstrom', format='fits')
WARNING: UnitsWarning: The unit 'Angstrom' has been deprecated
in the FITS standard. Suggested: nm (with data multiplied by
0.1).  [astropy.units.format.utils]
```

astropy.utils

- `treat_deprecations_as_exceptions` has been fixed to recognize Astropy deprecation warnings. [#3015]
- Converted representation of progress bar units without suffix from float to int in console.human_file_size. [#2201, #2202, #2721, #3299]

astropy.wcs

- `astropy.wcs.WCS.sub` now accepts unicode strings as input on Python 2.x [#3356]

Misc

- Some modules and tests that would crash upon import when using a non-final release of Numpy (e.g. 1.9.0rc1). [#3471]

Other Changes and Additions

- The bundled copy of astropy-helpers has been updated to v1.0. [#3515]
- Updated `astropy.extern.configobj` to Version 5. Version 5 uses `six` and the same code covers both Python 2 and Python 3. [#3149]

astropy.coordinates

- The `repr` of `SkyCoord` and coordinate frame classes now separate frame attributes and coordinate information. [#2704, #2882]

astropy.io.fits

- Overwriting an existing file using the `clobber=True` option no longer displays a warning message. [#1963]
- `fits.open` no longer catches `OSError` exceptions on missing or unreadable files– instead it raises the standard Python exceptions in such cases. [#2756, #2785]

astropy.table

- Sped up setting of `Column` slices by an order of magnitude. [#2994, #3020]
- Updated the bundled `six` module to version 1.7.3 and made 1.7.3 the minimum acceptable version of `six`. [#2814]
- The version of ERFA included with Astropy is now v1.1.1 [#2971]
- The code base is now fully Python 2 and 3 compatible and no longer requires 2to3. [#2033]
- funcsigs is included in utils.compat, but defaults to the inspect module components where available (3.3+) [#3151].
- The list of modules displayed in the pytest header can now be customized. [#3157]
- jinja2>=2.7 is now required to build the source code from the git repository, in order to allow the ERFA wrappers to be generated. [#3166]

0.4.5 (2015-02-16)

Bug Fixes

- Fixed unnecessary attempt to run `git` when importing astropy. In particular, fixed a crash in Python 3 that could result from this when importing Astropy when the the current working directory is an empty git repository. [#3475]

Other Changes and Additions

- Updated bundled copy of astropy-helpers to v0.4.6. [#3508]

0.4.4 (2015-01-21)

Bug Fixes

astropy.vo.samp

- `astropy.vo.samp` is now usable on Python builds that do not support the SSLv3 protocol (which depends both on the version of Python and the version of OpenSSL or LibreSSL that it is built against.) [#3308]

API Changes

astropy.vo.samp

- The default SSL protocol used is now determined from the default used in the Python `ssl` standard library. This default may be different depending on the exact version of Python you are using. [#3308]

astropy.wcs

- WCS allows slices of the form slice(None, x, y), which previously resulted in an unsliced copy being returned (note: this was previously incorrectly reported as fixed in v0.4.3) [#2909]

0.4.3 (2015-01-15)

Bug Fixes

astropy.coordinates

- The `Distance` class has been fixed to no longer rely on the deprecated cosmology functions. [#2991]
- Ensure `float32` values can be used in coordinate representations. [#2983]
- Fix frame attribute inheritance in `SkyCoord.transform_to()` method so that the default attribute value (e.g. equinox) for the destination frame gets used if no corresponding value was explicitly specified. [#3106]

- `Angle` accepts hours:mins or deg:mins initializers (without seconds). In these cases float minutes are also accepted. [#2843]
- `astropy.coordinates.SkyCoord` objects are now copyable. [#2888]
- `astropy.coordinates.SkyCoord` object attributes are now immutable. It is still technically possible to change the internal data for an array-valued coordinate object but this leads to inconsistencies [#2889] and should not be done. [#2888]

astropy.cosmology

- The `ztol` keyword argument to z_at_value now works correctly [#2993].

astropy.io.ascii

- Fix a bug in Python 3 when guessing file format using a file object as input. Also improve performance in same situation for Python 2. [#3132]
- Fix a problem where URL was being downloaded for each guess. [#2001]

astropy.io.fits

- The `in` operator now works correctly for checking if an extension is in an `HDUList` (as given via EXTNAME, (EXTNAME, EXTVER) tuples, etc.) [#3060]
- Added workaround for bug in MacOS X <= 10.8 that caused np.fromfile to fail. [#3078]
- Added support for the `RICE_ONE` compression type synonym. [#3115]

astropy.modeling

- Fixed a test failure on Debian/PowerPC and Debian/s390x. [#2708]
- Fixed crash in evaluating models that have more outputs than inputs–this case may not be handled as desired for all conceivable models of this format (some may have to implement custom `prepare_inputs` and `prepare_outputs` methods). But as long as all outputs can be assumed to have a shape determined from the broadcast of all inputs with all parameters then this can be used safely. [#3250]

astropy.table

- Fix a bug that caused join to fail for multi-dimensional columns. [#2984]
- Fix a bug where MaskedColumn attributes which had been changed since the object was created were not being carried through when slicing. [#3023]
- Fix a bug that prevented initializing a table from a structured array with multi-dimensional columns with copy=True. [#3034]
- Fixed unnecessarily large unicode columns when instantiating a table from

row data on Python 3. [#3052]
- Improved the warning message when unable to aggregate non-numeric columns. [#2700]

astropy.units

- Operations on quantities with incompatible types now raises a much more informative `TypeError`. [#2934]
- `Quantity.tolist` now overrides the `ndarray` method to give a `NotImplementedError` (by renaming the previous `list` method). [#3050]
- `Quantity.round` now always returns a `Quantity` (previously it returned an `ndarray` for `decimals>0`). [#3062]
- Ensured `np.squeeze` always returns a `Quantity` (it only worked if no dimensions were removed). [#3045]
- Input to `Quantity` with a `unit` attribute no longer can get mangled with `copy=False`. [#3051]
- Remove trailing space in `__format__` calls for dimensionless quantities. [#3097]
- Comparisons between units and non-unit-like objects now works correctly. [#3108]
- Units with fractional powers are now correctly multiplied together by using rational arithmetic. [#3121]
- Removed a few entries from spectral density equivalencies which did not make sense. [#3153]

astropy.utils

- Fixed an issue with the `deprecated` decorator on classes that invoke `super()` in their `__init__` method. [#3004]
- Fixed a bug which caused the `metadata_conflicts` parameter to be ignored in the `astropy.utils.metadata.merge` function. [#3294]

astropy.vo

- Fixed an issue with reconnecting to a SAMP Hub. [#2674]

astropy.wcs

- Invalid or out of range values passed to `wcs_world2pix` will now be correctly identified and returned as `nan` values. [#2965]
- Fixed an issue which meant that Python thought `WCS` objects were iterable. [#3066]

Misc

- Astropy will now work if your Python interpreter does not have the `bz2` module installed. [#3104]
- Fixed `ResourceWarning` for `astropy/extern/bundled/six.py` that could occur sometimes after using Astropy in Python 3.4. [#3156]

Other Changes and Additions

astropy.coordinates

- Improved the agreement of the FK5 <-> Galactic conversion with other codes, and with the FK5 <-> FK4 <-> Galactic route. [#3107]

0.4.2 (2014-09-23)

Bug Fixes

astropy.coordinates

- `Angle` accepts hours:mins or deg:mins initializers (without seconds). In these cases float minutes are also accepted.
- The `repr` for coordinate frames now displays the frame attributes (ex: ra, dec) in a consistent order. It should be noted that as part of this fix, the `BaseCoordinateFrame.get_frame_attr_names()` method now returns an `OrderedDict` instead of just a `dict`. [#2845]

astropy.io.fits

- Fixed a crash when reading scaled float data out of a FITS file that was loaded from a string (using `HDUList.fromfile`) rather than from a file. [#2710]
- Fixed a crash when reading data from an HDU whose header contained in invalid value for the BLANK keyword (e.g., a string value instead of an integer as required by the FITS Standard). Invalid BLANK keywords are now warned about, but are otherwise ignored. [#2711]
- Fixed a crash when reading the header of a tile-compressed HDU if that header contained invalid duplicate keywords resulting in a `KeyError` [#2750]
- Fixed crash when reading gzip-compressed FITS tables through the Astropy `Table` interface. [#2783]
- Fixed corruption when writing new FITS files through to gzipped files. [#2794]
- Fixed crash when writing HDUs made with non-contiguous data arrays to file-like objects. [#2794]

- It is now possible to create `astropy.io.fits.BinTableHDU` objects with a table with zero rows. [#2916]

astropy.io.misc

- Fixed a bug that prevented h5py `Dataset` objects from being automatically recognized by `Table.read`. [#2831]

astropy.modeling

- Make `LevMarLSQFitter` work with `weights` keyword. [#2900]

astropy.table

- Fixed reference cycle in tables that could prevent `Table` objects from being freed from memory. [#2879]
- Fixed an issue where `Table.pprint()` did not print the header to `stdout` when `stdout` is redirected (say, to a file). [#2878]
- Fixed printing of masked values when a format is specified. [#1026]
- Ensured that numpy ufuncs that return booleans return plain `ndarray` instances, just like the comparison operators. [#2963]

astropy.time

- Ensure bigendian input to Time works on a little-endian machine (and vice versa). [#2942]

astropy.units

- Ensure unit is kept when adding 0 to quantities. [#2968]

astropy.utils

- Fixed color printing on Windows with IPython 2.0. [#2878]

astropy.vo

- Improved error message on Cone Search time out. [#2687]

Other Changes and Additions

- Fixed a couple issues with files being inappropriately included and/or excluded from the source archive distributions of Astropy. [#2843, #2854]
- As part of fixing the fact that masked elements of table columns could not be printed when a format was specified, the column format string options were expanded to allow simple specifiers such as `'5.2f'`. [#2898]
- Ensure numpy 1.9 is supported. [#2917]

- Ensure numpy master is supported, by making `np.cbrt` work with quantities. [#2937]

0.4.1 (2014-08-08)

Bug Fixes

astropy.config

- Fixed a bug where an unedited configuration file from astropy 0.3.2 would not be correctly identified as unedited. [#2772] This resulted in the warning:

```
WARNING: ConfigurationChangedWarning: The configuration options
in astropy 0.4 may have changed, your configuration file was not
updated in order to preserve local changes.  A new configuration
template has been saved to
'~/.astropy/config/astropy.0.4.cfg'. [astropy.config.configuration]
```

- Fixed the error message that is displayed when an old configuration item has moved. Before, the destination section was wrong. [#2772]
- Added configuration settings for `io.fits`, `io.votable` and `table.jsviewer` that were missing from the configuration file template. [#2772]
- The configuration template is no longer rewritten on every import of astropy, causing race conditions. [#2805]

astropy.convolution

- Fixed the multiplication of `Kernel` with numpy floats. [#2174]

astropy.coordinates

- `Distance` can now take a list of quantities. [#2261]
- For in-place operations for `Angle` instances in which the result unit is not an angle, an exception is raised before the instance is corrupted. [#2718]
- `CartesianPoints` are now deprecated in favor of `CartesianRepresentation`. [#2727]

astropy.io.misc

- An existing table within an HDF5 file can be overwritten without affecting other datasets in the same HDF5 file by simultaneously using `overwrite=True` and `append=True` arguments to the `Table.write` method. [#2624]

astropy.logger

- Fixed a crash that could occur in rare cases when (such as in bundled apps) where submodules of the `email` package are not importable. [#2671]

astropy.nddata

- `astropy.nddata.NDData()` no longer raises a `ValueError` when passed a numpy masked array which has no masked entries. [#2784]

astropy.table

- When saving a table to a FITS file containing a unit that is not supported by the FITS standard, a warning rather than an exception is raised. [#2797]

astropy.units

- By default, `Quantity` and its subclasses will now convert to float also numerical types such as `decimal.Decimal`, which are stored as objects by numpy. [#1419]
- The units `count`, `pixel`, `voxel` and `dbyte` now output to FITS, OGIP and VOUnit formats correctly. [#2798]

astropy.utils

- Restored missing information from deprecation warning messages from the `deprecated` decorator. [#2811]
- Fixed support for `staticmethod` deprecation in the `deprecated` decorator. [#2811]

astropy.wcs

- Fixed a memory leak when `astropy.wcs.WCS` objects are copied [#2754]
- Fixed a crash when passing `ra_dec_order=True` to any of the `*2world` methods. [#2791]

Other Changes and Additions

- Bundled copy of astropy-helpers upgraded to v0.4.1. [#2825]
- General improvements to documentation and docstrings [#2722, #2728, #2742]
- Made it easier for third-party packagers to have Astropy use their own version of the `six` module (so long as it meets the minimum version requirement) and remove the copy bundled with Astropy. See the astropy/extern/README file in the source tree. [#2623]

0.4 (2014-07-16)

New Features

astropy.constants

- Added `b_wien` to represent Wien wavelength displacement law constant. [#2194]

astropy.convolution

- Changed the input parameter in `Gaussian1DKernel` and `Gaussian2DKernel` from `width` to `stddev` [#2085].

astropy.coordinates

- The coordinates package has undergone major changes to implement APE5 . These include backwards-incompatible changes, as the underlying framework has changed substantially. See the APE5 text and the package documentation for more details. [#2422]
- A `position_angle` method has been added to the new `SkyCoord`. [#2487]
- Updated `Angle.dms` and `Angle.hms` to return `namedtuple` -s instead of regular tuples, and added `Angle.signed_dms` attribute that gives the absolute value of the `d`, `m`, and `s` along with the sign. [#1988]
- By default, `Distance` objects are now required to be positive. To allow negative values, set `allow_negative=True` in the `Distance` constructor when creating a `Distance` instance.
- `Longitude` (resp. `Latitude`) objects cannot be used any more to initialize or set `Latitude` (resp. `Longitude`) objects. An explicit conversion to `Angle` is now required. [#2461]
- The deprecated functions for pre-0.3 coordinate object names like `ICRSCoordinates` have been removed. [#2422]
- The `rotation_matrix` and `angle_axis` functions in `astropy.coordinates.angles` were made more numerically consistent and are now tested explicitly [#2619]

astropy.cosmology

- Added `z_at_value` function to find the redshift at which a cosmology function matches a desired value. [#1909]
- Added `FLRW.differential_comoving_volume` method to give the differential comoving volume at redshift z. [#2103]

- The functional interface is now deprecated in favor of the more-explicit use of methods on cosmology objects. [#2343]
- Updated documentation to reflect the removal of the functional interface. [#2507]

astropy.io.ascii

- The `astropy.io.ascii` output formats `latex` and `aastex` accept a dictionary called `latex_dict` to specify options for LaTeX output. It is now possible to specify the table alignment within the text via the `tablealign` keyword. [#1838]
- If `header_start` is specified in a call to `ascii.get_reader` or any method that calls `get_reader` (e.g. `ascii.read`) but `data_start` is not specified at the same time, then `data_start` is calculated so that the data starts after the header. Before this, the default was that the header line was read again as the first data line [#855 and #1844].
- A new `csv` format was added as a convenience for handling CSV (comma-separated values) data. [#1935] This format also recognises rows with an inconsistent number of elements. [#1562]
- An option was added to guess the start of data for CDS format files when they do not strictly conform to the format standard. [#2241]
- Added an HTML reader and writer to the `astropy.io.ascii` package. Parsing requires the installation of BeautifulSoup and is therefore an optional feature. [#2160]
- Added support for inputting column descriptions and column units with the `io.ascii.SExtractor` reader. [#2372]
- Allow the use of non-local ReadMe files in the CDS reader. [#2329]
- Provide a mechanism to select how masked values are printed. [#2424]
- Added support for reading multi-aperture daophot file. [#2656]

astropy.io.fits

- Included a new command-line script called `fitsheader` to display the header(s) of a FITS file from the command line. [#2092]
- Added new verification options `fix+ignore`, `fix+warn`, `fix+exception`, `silentfix+ignore`, `silentfix+warn`, and `silentfix+exception` which give more control over how to report fixable errors as opposed to unfixable errors.

astropy.modeling

- Prototype implementation of fitters that treat optimization algorithms separately from fit statistics, allowing new fitters to be created by mixing and matching optimizers and statistic functions. [#1914]

- Slight overhaul to how inputs to and outputs from models are handled with respect to array-valued parameters and variables, as well as sets of multiple models. See the associated PR and the modeling section of the v0.4 documentation for more details. [#2634]
- Added a new `SimplexLSQFitter` which uses a downhill simplex optimizer with a least squares statistic. [#1914]
- Changed `Gaussian2D` model such that `theta` now increases counterclockwise. [#2199]
- Replaced the `MatrixRotation2D` model with a new model called simply `Rotation2D` which requires only an angle to specify the rotation. The new `Rotation2D` rotates in a counter-clockwise sense whereas the old `MatrixRotation2D` increased the angle clockwise. [#2266, #2269]
- Added a new `AffineTransformation2D` model which serves as a replacement for the capability of `MatrixRotation2D` to accept an arbitrary matrix, while also adding a translation capability. [#2269]
- Added `GaussianAbsorption1D` model. [#2215]
- New `Redshift` model [#2176].

astropy.nddata

- Allow initialization `NDData` or `StdDevUncertainty` with a `Quantity`. [#2380]

astropy.stats

- Added flat prior to binom_conf_interval and binned_binom_proportion
- Change default in `sigma_clip` from `np.median` to `np.ma.median`. [#2582]

astropy.sphinx

- Note, the following new features are included in astropy-helpers as well:
- The `automodapi` and `automodsumm` extensions now include sphinx configuration options to write out what `automodapi` and `automodsumm` generate, mainly for debugging purposes. [#1975, #2022]
- Reference documentation now shows functions/class docstrings at the inteded user-facing API location rather than the actual file where the implementation is found. [#1826]
- The `automodsumm` extension configuration was changed to generate documentation of class `__call__` member functions. [#1817, #2135]
- `automodapi` and `automodsumm` now have an `:allowed-package-names:` option that make it possible to document functions and classes that are in a different namespace. [#2370]

astropy.table

- Improved grouped table aggregation by using the numpy `reduceat()` method when possible. This can speed up the operation by a factor of at least 10 to 100 for large unmasked tables and columns with relatively small group sizes. [#2625]
- Allow row-oriented data input using a new `rows` keyword argument. [#850]
- Allow subclassing of `Table` and the component classes `Row`, `Column`, `MaskedColumn`, `TableColumns`, and `TableFormatter`. [#2287]
- Fix to allow numpy integer types as valid indices into tables in Python 3.x [#2477]
- Remove transition code related to the order change in `Column` and `MaskedColumn` arguments `name` and `data` from Astropy 0.2 to 0.3. [#2511]
- Change HTML table representation in IPython notebook to show all table columns instead of restricting to 80 column width. [#2651]

astropy.time

- Mean and apparent sidereal time can now be calculated using the `sidereal_time` method [#1418].
- The time scale now defaults to UTC if no scale is provided. [#2091]
- `TimeDelta` objects can have all scales but UTC, as well as, for consistency with time-like quantities, undefined scale (where the scale is taken from the object one adds to or subtracts from). This allows, e.g., to work consistently in TDB. [#1932]
- `Time` now supports ISO format strings that end in “Z”. [#2211, #2203]

astropy.units

- Support for the unit format Office of Guest Investigator Programs (OGIP) FITS files has been added. [#377]
- The `spectral` equivalency can now handle angular wave number. [#1306 and #1899]
- Added `one` as a shorthand for `dimensionless_unscaled`. [#1980]
- Added `dex` and `dB` units. [#1628]
- Added `temperature()` equivalencies to support conversion between Kelvin, Celsius, and Fahrenheit. [#2209]
- Added `temperature_energy()` equivalencies to support conversion between electron-volt and Kelvin. [#2637]
- The runtime of `astropy.units.Unit.compose` is greatly improved (by a factor of 2 in most cases) [#2544]
- Added `electron` unit. [#2599]

astropy.utils

- `timer.RunTimePredictor` now uses `astropy.modeling` in its `do_fit()` method. [#1896]

astropy.vo

- A new sub-package, `astropy.vo.samp`, is now available (this was previously the SAMPy package, which has been refactored for use in Astropy). [#1907]
- Enhanced functionalities for `VOSCatalog` and `VOSDatabase`. [#1206]

astropy.wcs

- astropy now requires wcslib version 4.23. The version of wcslib included with astropy has been updated to version 4.23.
- Bounds checking is now performed on native spherical coordinates. Any out-of-bounds values will be returned as `NaN`, and marked in the `stat` array, if using the low-level `wcslib` interface such as `astropy.wcs.Wcsprm.p2s`. [#2107]
- A new method, `astropy.wcs.WCS.compare()`, compares two wcsprm structs for equality with varying degrees of strictness. [#2361]
- New `astropy.wcs.utils` module, with a handful of tools for manipulating WCS objects, including dropping, swapping, and adding axes.

Misc

- Includes the new astropy-helpers package which separates some of Astropy's build, installation, and documentation infrastructure out into an independent package, making it easier for Affiliated Packages to depend on these features. astropy-helpers replaces/deprecates some of the submodules in the `astropy` package (see API Changes below). See also APE 4 for more details on the motivation behind and implementation of astropy-helpers. [#1563]

API Changes

astropy.config

- The configuration system received a major overhaul, as part of APE3. It is no longer possible to save configuration items from Python, but instead users must edit the configuration file directly. The locations of configuration items have moved, and some have been changed to science state values. The old locations should continue to work until astropy 0.5, but deprecation warnings will be displayed. See the Configuration transition docs for a detailed description of the changes and how to update existing code. [#2094]

astropy.io.fits

- The `astropy.io.fits.new_table` function is now fully deprecated (though will not be removed for a long time, considering how widely it is used).

  Instead please use the more explicit `BinTableHDU.from_columns` to create a new binary table HDU, and the similar `TableHDU.from_columns` to create a new ASCII table. These otherwise accept the same arguments as `new_table` which is now just a wrapper for these.
- The `.fromstring` classmethod of each HDU type has been simplified such that, true to its namesake, it only initializes an HDU from a string containing its header *and* data.
- Fixed an issue where header wildcard matching (for example `header['DATE*']`) can be used to match *any* characters that might appear in a keyword. Previously this only matched keywords containing characters in the set `[0-9A-Za-z_]`. Now this can also match a hyphen `-` and any other characters, as some conventions like `HIERARCH` and record-valued keyword cards allow a wider range of valid characters than standard FITS keywords.
- This will be the *last* release to support the following APIs that have been marked deprecated since Astropy v0.1/PyFITS v3.1:
- The `CardList` class, which was part of the old header implementation.
- The `Card.key` attribute. Use `Card.keyword` instead.
- The `Card.cardimage` and `Card.ascardimage` attributes. Use simply `Card.image` or `str(card)` instead.
- The `create_card` factory function. Simply use the normal `Card` constructor instead.
- The `create_card_from_string` factory function. Use `Card.fromstring` instead.
- The `upper_key` function. Use `Card.normalize_keyword` method instead (this is not unlikely to be used outside of PyFITS itself, but it was technically public API).
- The usage of `Header.update` with `Header.update(keyword, value, comment)` arguments. `Header.update` should only be used analogously to `dict.update`. Use `Header.set` instead.
- The `Header.ascard` attribute. Use `Header.cards` instead for a list of

all the `Card` objects in the header.

- The `Header.rename_key` method. Use `Header.rename_keyword` instead.
- The `Header.get_history` method. Use `header['HISTORY']` instead (normal keyword lookup).
- The `Header.get_comment` method. Use `header['COMMENT']` instead.
- The `Header.toTxtFile` method. Use `header.totextfile` instead.
- The `Header.fromTxtFile` method. Use `Header.fromtextfile` instead.
- The `tdump` and `tcreate` functions. Use `tabledump` and `tableload` respectively.
- The `BinTableHDU.tdump` and `tcreate` methods. Use `BinTableHDU.dump` and `BinTableHDU.load` respectively.
- The `txtfile` argument to the `Header` constructor. Use `Header.fromfile` instead.
- The `startColumn` and `endColumn` arguments to the `FITS_record` constructor. These are unlikely to be used by any user code.

  These deprecated interfaces will be removed from the development version of Astropy following the v0.4 release (they will still be available in any v0.4.x bugfix releases, however).

astropy.modeling

- The method computing the derivative of the model with respect to parameters was renamed from `deriv` to `fit_deriv`. [#1739]
- `ParametricModel` and the associated `Parametric1DModel` and `Parametric2DModel` classes have been renamed `FittableModel`, `Fittable1DModel`, and `Fittable2DModel` respectively. The base `Model` class has subsumed the functionality of the old

  `ParametricModel` class so that all models support parameter constraints. The only distinction of `FittableModel` is that anything which subclasses it is assumed “safe” to use with Astropy fitters. [#2276]
- `NonLinearLSQFitter` has been renamed `LevMarLSQFitter` to emphasise that it uses the Levenberg-Marquardt optimization algorithm with a least squares statistic function. [#1914]
- The `SLSQPFitter` class has been renamed `SLSQPLSQFitter` to emphasize that it uses the Sequential Least Squares Programming

optimization algorithm with a least squares statistic function. [#1914]

- The `Fitter.errorfunc` method has been renamed to the more general `Fitter.objective_function`. [#1914]

astropy.nddata

- Issue warning if unit is changed from a non-trivial value by directly setting `NDData.unit`. [#2411]
- The `mask` and `flag` attributes of `astropy.nddata.NDData` can now be set with any array-like object instead of requiring that they be set with a `numpy.ndarray`. [#2419]

astropy.sphinx

- Use of the `astropy.sphinx` module is deprecated; all new development of this module is in `astropy_helpers.sphinx` which should be used instead (therefore documentation builds that made use of any of the utilities in `astropy.sphinx` now have `astropy_helpers` as a documentation dependency).

astropy.table

- The default table printing function now shows a table header row for units if any columns have the unit attribute set. [#1282]
- Before, an unmasked `Table` was automatically converted to a masked table if generated from a masked Table or a `MaskedColumn`. Now, this conversion is only done if explicitly requested or if any of the input values is actually masked. [#1185]
- The repr() function of `astropy.table.Table` now shows the units if any columns have the unit attribute set. [#2180]
- The semantics of the config options `table.max_lines` and `table.max_width` has changed slightly. If these values are not set in the config file, astropy will try to determine the size automatically from the terminal. [#2683]

astropy.time

- Correct use of UT in TDB calculation [#1938, #1939].
- `TimeDelta` objects can have scales other than TAI [#1932].
- Location information should now be passed on via an `EarthLocation` instance or anything that initialises it, e.g., a tuple containing either geocentric or geodetic coordinates. [#1928]

astropy.units

- `Quantity` now converts input to float by default, as this is physically most sensible for nearly all units [#1776].
- `Quantity` comparisons with `==` or `!=` now always return `True` or `False`, even if units do not match (for which case a `UnitsError` used to be raised). [#2328]
- Applying `float` or `int` to a `Quantity` now works for all dimensionless quantities; they are automatically converted to unscaled dimensionless. [#2249]
- The exception `astropy.units.UnitException`, which was deprecated in astropy 0.2, has been removed. Use `astropy.units.UnitError` instead [#2386]
- Initializing a `Quantity` with a valid number/array with a `unit` attribute now interprets that attribute as the units of the input value. This makes it possible to initialize a `Quantity` from an Astropy `Table` column and have it correctly pick up the units from the column. [#2486]

astropy.wcs

- `calcFootprint` was deprecated. It is replaced by `calc_footprint`. An optional boolean keyword `center` was added to `calc_footprint`. It controls whether the centers or the corners of the pixels are used in the computation. [#2384]
- `astropy.wcs.WCS.sip_pix2foc` and `astropy.wcs.WCS.sip_foc2pix` formerly did not conform to the `SIP` standard: `CRPIX` was added to the `foc` result so that it could be used as input to “core FITS WCS”. As of astropy 0.4, `CRPIX` is no longer added to the result, so the `foc` space is correct as defined in the SIP convention. [#2360]
- `astropy.wcs.UnitConverter`, which was deprecated in astropy 0.2, has been removed. Use the `astropy.units` module instead. [#2386]
- The following methods on `astropy.wcs.WCS`, which were deprecated in astropy 0.1, have been removed [#2386]:
- `all_pix2sky` -> `all_pix2world`
- `wcs_pix2sky` -> `wcs_pix2world`
- `wcs_sky2pix` -> `wcs_world2pix`
- The `naxis1` and `naxis2` attributes and the `get_naxis` method of `astropy.wcs.WCS`, which were deprecated in astropy 0.2, have been removed. Use the shape of the underlying FITS data array instead. [#2386]

Misc

- The `astropy.setup_helpers` and `astropy.version_helpers` modules are deprecated; any non-critical fixes and development to those modules should be in `astropy_helpers` instead. Packages that use these modules in their `setup.py` should depend on `astropy_helpers` following the same pattern as in the Astropy package template.

Bug Fixes

astropy.constants

- `astropy.constants.Contant` objects can now be deep copied. [#2601]

astropy.cosmology

- The distance modulus function in `astropy.cosmology` can now handle negative distances, which can occur in certain closed cosmologies. [#2008]
- Removed accidental imports of some extraneous variables in `astropy.cosmology` [#2025]

astropy.io.ascii

- `astropy.io.ascii.read` would fail to read lists of strings where some of the strings consisted of just a newline (“n”). [#2648]

astropy.io.fits

- Use NaN for missing values in FITS when using Table.write for float columns. Earlier the default fill value was close to 1e20.[#2186]
- Fixes for checksums on 32-bit platforms. Results may be different if writing or checking checksums in “nonstandard” mode. [#2484]
- Additional minor bug fixes ported from PyFITS. [#2575]

astropy.io.votable

- It is now possible to save an `astropy.table.Table` object as a VOTable with any of the supported data formats, `tabledata`, `binary` and `binary2`, by using the `tabledata_format` kwarg. [#2138]
- Fixed a crash writing out variable length arrays. [#2577]

astropy.nddata

- Indexing `NDData` in a way that results in a single element returns that element. [#2170]
- Change construction of result of arithmetic and unit conversion to allow subclasses to require the presence of attribute like unit. [#2300]
- Scale uncertainties to correct units in arithmetic operations and unit conversion. [#2393]

- Ensure uncertainty and mask members are copied in arithmetic and convert_unit_to. [#2394]
- Mask result of arithmetic if either of the operands is masked. [#2403]
- Copy all attributes of input object if `astropy.nddata.NDData` is initialized with an `NDData` object. [#2406]
- Copy `flags` to new object in `convert_unit_to`. [#2409]
- Result of `NDData` arithmetic makes a copy of any WCS instead of using a reference. [#2410]
- Fix unit handling for multiplication/division and use `astropy.units.Quantity` for units arithmetic. [#2413]
- A masked `NDData` is now converted to a masked array when used in an operation or ufunc with a numpy array. [#2414]
- An unmasked `NDData` now uses an internal representation of its mask state that `numpy.ma` expects so that an `NDData` behaves as an unmasked array. [#2417]

astropy.sphinx

- Fix crash in smart resolver when the resolution doesn’t work. [#2591]

astropy.table

- The `astropy.table.Column` object can now use both functions and callable objects as formats. [#2313]
- Fixed a problem on 64 bit windows that caused errors “expected ‘DTYPE_t’ but got ‘long long’” [#2490]
- Fix initialisation of `TableColumns` with lists or tuples. [#2647]
- Fix removal of single column using `remove_columns`. [#2699]
- Fix a problem that setting a row element within a masked table did not update the corresponding table element. [#2734]

astropy.time

- Correct UT1->UTC->UT1 round-trip being off by 1 second if UT1 is on a leap second. [#2077]

astropy.units

- `Quantity.copy` now behaves identically to `ndarray.copy`, and thus supports the `order` argument (for numpy >=1.6). [#2284]
- Composing base units into identical composite units now works. [#2382]
- Creating and composing/decomposing units is now substantially faster [#2544]
- `Quantity` objects now are able to be assigned NaN [#2695]

astropy.wcs

- Astropy now requires wcslib version 4.23. The version of wcslib included with astropy has been updated to version 4.23.
- Bug fixes in the projection routines: in `hpxx2s` [the cartesian-to-spherical operation of the `HPX` projection] relating to bounds checking, bug introduced at wcslib 4.20; in `parx2s` and molx2s`` [the cartesion-to-spherical operation of the `PAR` and `MOL` projections respectively] relating to setting the stat vector; in `hpxx2s` relating to implementation of the vector API; and in `xphx2s` relating to setting an out-of-bounds value of *phi*.
- In the `PCO` projection, use alternative projection equations for greater numerical precision near theta == 0. In the `COP` projection, return an exact result for theta at the poles. Relaxed the tolerance for bounds checking a little in `SFL` projection.
- Fix a bug allocating insufficient memory in `astropy.wcs.WCS.sub` [#2468]
- A new method, `Wcsprm.bounds_check` (corresponding to wcslib's `wcsbchk`) has been added to control what bounds checking is performed by wcslib.
- `WCS.to_header` will now raise a more meaningful exception when the WCS information is invalid or inconsistent in some way. [#1854]
- In `WCS.to_header`, `RESTFRQ` and `RESTWAV` are no longer rewritten if zero. [#2468]
- In `WCS.to_header`, floating point values will now always be written with an exponent or fractional part, i.e. `.0` being appended if necessary to acheive this. [#2468]
- If the C extension for `astropy.wcs` was not built or fails to import for any reason, `import astropy.wcs` will result in an `ImportError`, rather than getting obscure errors once the `astropy.wcs` is used. [#2061]
- When the C extension for `astropy.wcs` is built using a version of `wscslib` already present in the system, the package does not try to install `wcslib` headers under `astropy/wcs/include`. [#2536]
- Fixes an unresolved external symbol error in the `astropy.wcs._wcs` C extension on Microsoft Windows when built with a Microsoft compiler. [#2478]

Misc

- Running the test suite with `python setup.py test` now works if the path to the source contains spaces. [#2488]
- The version of ERFA included with Astropy is now v1.1.0 [#2497]
- Removed deprecated option from travis configuration and force use of

wheels rather than allowing build from source. [#2576]
- The short option `-n` to run tests in parallel was broken (conflicts with the distutils built-in option of “dry-run”). Changed to `-j`. [#2566]

Other Changes and Additions

- python setup.py test –coverage will now give more accurate results, because the coverage analysis will include early imports of astropy. There doesn’t seem to be a way to get this to work when doing `import astropy; astropy.test()`, so the `coverage` keyword to `astropy.test` has been removed. Coverage testing now depends only on coverage.py, not `pytest-cov`. [#2112]
- The included version of py.test has been upgraded to 2.5.1. [#1970]
- The included version of six.py has been upgraded to 1.5.2. [#2006]
- Where appropriate, tests are now run both with and without the `unicode_literals` option to ensure that we support both cases. [#1962]
- Running the Astropy test suite from within the IPython REPL is disabled for now due to bad interaction between the test runner and IPython’s logging and I/O handler. For now, run the Astropy tests should be run in the basic Python interpreter. [#2684]
- Added support for numerical comparison of floating point values appearing in the output of doctests using a `+FLOAT_CMP` doctest flag. [#2087]
- A monkey patch is performed to fix a bug in Numpy version 1.7 and earlier where unicode fill values on masked arrays are not supported. This may cause unintended side effects if your application also monkey patches `numpy.ma` or relies on the broken behavior. If unicode support of masked arrays is important to your application, upgrade to Numpy 1.8 or later for best results. [#2059]
- The developer documentation has been extensively rearranged and rewritten. [#1712]
- The `human_time` function in `astropy.utils` now returns strings without zero padding. [#2420]
- The `bdist_dmg` command for `setup.py` has now been removed. [#2553]
- Many broken API links have been fixed in the documentation, and the `nitpick` Sphinx option is now used to avoid broken links in future. [#1221, #2019, #2109, #2161, #2162, #2192, #2200, #2296, #2448, #2456, #2460, #2467, #2476, #2508, #2509]

0.3.2 (2014-05-13)

Bug Fixes

astropy.coordinates

- if `sep` argument is specified to be a single character in `sexagisimal_to_string`, it now includes seperators only between items [#2183]
- Ensure comparisons involving `Distance` objects do not raise exceptions; also ensure operations that lead to units other than length return `Quantity`. [#2206, #2250]
- Multiplication and division of `Angle` objects is now supported. [#2273]
- Fixed `Angle.to_string` functionality so that negative angles have the correct amount of padding when `pad=True`. [#2337]
- Mixing strings and quantities in the `Angle` constructor now works. For example: `Angle(['1d', 1. * u.d])`. [#2398]
- If `Longitude` is given a `Longitude` as input, use its `wrap_angle` by default [#2705]

astropy.cosmology

- Fixed `format()` compatibility with Python 2.6. [#2129]
- Be more careful about converting to floating point internally [#1815, #1818]

astropy.io.ascii

- The CDS reader in `astropy.io.ascii` can now handle multiple description lines in ReadMe files. [#2225]
- When reading a table with values that generate an overflow error during type conversion (e.g. overflowing the native C long type), fall through to using string. Previously this generated an exception [#2234].
- Recognize any string with one to four dashes as null value. [#1335]

astropy.io.fits

- Allow pickling of `FITS_rec` objects. [#1597]
- Improved behavior when writing large compressed images on OSX by removing an unnecessary check for platform architecture. [#2345]
- Fixed an issue where Astropy `Table` objects containing boolean columns were not correctly written out to FITS files. [#1953]
- Several other bug fixes ported from PyFITS v3.2.3 [#2368]
- Fixed a crash on Python 2.x when writing a FITS file directly to a `StringIO.StringIO` object. [#2463]

astropy.io.registry

- Allow readers/writers with the same name to be attached to different classes.

[#2312]

astropy.io.votable

- By default, floating point values are now written out using `repr` rather than `str` to preserve precision [#2137]

astropy.modeling

- Fixed the `SIP` and `InverseSIP` models both so that they work in the first place, and so that they return results consistent with the SIP functions in `astropy.wcs`. [#2177]

astropy.stats

- Ensure the `axis` keyword in `astropy.stats.funcs` can now be used for all axes. [#2173]

astropy.table

- Ensure nameless columns can be printed, using 'None' for the header. [#2213]

astropy.time

- Fixed pickling of `Time` objects. [#2123]

astropy.units

- `Quantity._repr_latex_()` returns `NotImplementedError` for quantity arrays instead of an uninformative formatting exception. [#2258]
- Ensure `Quantity.flat` always returns `Quantity`. [#2251]
- Angstrom unit renders better in MathJax [#2286]

astropy.utils

- Progress bars will now be displayed inside the IPython qtconsole. [#2230]
- `data.download_file()` now evaluates `REMOTE_TIMEOUT()` at runtime rather than import time. Previously, setting `REMOTE_TIMEOUT` after import had no effect on the function's behavior. [#2302]
- Progressbar will be limited to 100% so that the bar does not exceed the terminal width. The numerical display can still exceed 100%, however.

astropy.vo

- Fixed `format()` compatibility with Python 2.6. [#2129]
- Cone Search validation no longer raises `ConeSearchError` for positive

RA. [#2240, #2242]

astropy.wcs

- Fixed a bug where calling `astropy.wcs.Wcsprm.sub` with `WCSSUB_CELESTIAL` may cause memory corruption due to underallocation of a temporary buffer. [#2350]
- Fixed a memory allocation bug in `astropy.wcs.Wcsprm.sub` and `astropy.wcs.Wcsprm.copy`. [#2439]

Misc

- Fixes for compatibility with Python 3.4. [#1945]
- `import astropy; astropy.test()` now correctly uses the same test configuration as `python setup.py test` [#1811]

0.3.1 (2014-03-04)

Bug Fixes

astropy.config

- Fixed a bug where `ConfigurationItem.set_temp()` does not reset to default value when exception is raised within `with` block. [#2117]

astropy.convolution

- Fixed a bug where `_truncation` was left undefined for `CustomKernel`. [#2016]
- Fixed a bug with `_normalization` when `CustomKernel` input array sums to zero. [#2016]

astropy.coordinates

- Fixed a bug where using `==` on two array coordinates wouldn’t work. [#1832]
- Fixed bug which caused `len()` not to work for coordinate objects and added a `.shape` property to get appropriately array-like behavior. [#1761, #2014]
- Fixed a bug where sexagesimal notation would sometimes include exponential notation in the last field. [#1908, #1913]
- `CompositeStaticMatrixTransform` no longer attempts to reference the undefined variable `self.matrix` during instantiation. [#1944]
- Fixed pickling of `Longitude`, ensuring `wrap_angle` is preserved [#1961]

- Allow `sep` argument in `Angle.to_string` to be empty (resulting in no separators) [#1989]

astropy.io.ascii

- Allow passing unicode delimiters when reading or writing tables. The delimiter must be convertible to pure ASCII. [#1949]
- Fix a problem when reading a table and renaming the columns to names that already exist. [#1991]

astropy.io.fits

- Ported all bug fixes from PyFITS 3.2.1. See the PyFITS changelog at https://pyfits.readthedocs.io/en/v3.2.1/ [#2056]

astropy.io.misc

- Fixed issues in the HDF5 Table reader/writer functions that occurred on Windows. [#2099]

astropy.io.votable

- The `write_null_values` kwarg to `VOTable.to_xml`, when set to **`False`** (the default) would produce non-standard VOTable files. Therefore, this functionality has been replaced by a better understanding that knows which fields in a VOTable may be left empty (only `char`, `float` and `double` in VOTable 1.1 and 1.2, and all fields in VOTable 1.3). The kwarg is still accepted but it will be ignored, and a warning is emitted. [#1809]
- Printing out a `astropy.io.votable.tree.Table` object using **`repr`** or **`str`** now uses the pretty formatting in `astropy.table`, so it's possible to easily preview the contents of a `VOTable`. [#1766]

astropy.modeling

- Fixed bug in computation of model derivatives in `LinearLSQFitter`. [#1903]
- Raise a `NotImplementedError` when fitting composite models. [#1915]
- Fixed bug in the computation of the `Gaussian2D` model. [#2038]
- Fixed bug in the computation of the `AiryDisk2D` model. [#2093]

astropy.sphinx

- Added slightly more useful debug info for AstropyAutosummary. [#2024]

astropy.table

- The column string representation for n-dimensional cells with only one element has been fixed. [#1522]
- Fix a problem that caused `MaskedColumn.__getitem__` to not preserve column metadata. [#1471, #1872]
- With Numpy prior to version 1.6.2, tables with Unicode columns now sort correctly. [#1867]
- `astropy.table` can now print out tables with Unicode columns containing non-ascii characters. [#1864]
- Columns can now be named with Unicode strings, as long as they contain only ascii characters. This makes using `astropy.table` easier on Python 2 when `from __future__ import unicode_literals` is used. [#1864]
- Allow pickling of `Table`, `Column`, and `MaskedColumn` objects. [#792]
- Fix a problem where it was not possible to rename columns after sorting or adding a row. [#2039]

astropy.time

- Fix a problem where scale conversion problem in TimeFromEpoch was not showing a useful error [#2046]
- Fix a problem when converting to one of the formats `unix`, `cxcsec`, `gps` or `plot_date` when the time scale is `UT1`, `TDB` or `TCB` [#1732]
- Ensure that `delta_ut1_utc` gets calculated when accessed directly, instead of failing and giving a rather obscure error message [#1925]
- Fix a bug when computing the TDB to TT offset. The transform routine was using meters instead of kilometers for the Earth vector. [#1929]
- Increase `__array_priority__` so that `TimeDelta` can convert itself to a `Quantity` also in reverse operations [#1940]
- Correct hop list from TCG to TDB to ensure that conversion is possible [#2074]

astropy.units

- `Quantity` initialisation rewritten for speed [#1775]
- Fixed minor string formatting issue for dimensionless quantities. [#1772]
- Fix error for inplace operations on non-contiguous quantities [#1834].
- The definition of the unit `bar` has been corrected to “1e5 Pascal” from “100 Pascal” [#1910]
- For units that are close to known units, but not quite, for example due to differences in case, the exception will now include recommendations. [#1870]
- The generic and FITS unit parsers now accept multiple slashes in the unit string. There are multiple ways to interpret them, but the approach taken

here is to convert “m/s/kg” to “m s-1 kg-1”. Multiple slashes are accepted, but discouraged, by the FITS standard, due to the ambiguity of parsing, so a warning is raised when it is encountered. [#1911]

- The use of “angstrom” (with a lower case “a”) is now accepted in FITS unit strings, since it is in common usage. However, since it is not officially part of the FITS standard, a warning will be issued when it is encountered. [#1911]
- Pickling unrecognized units will not raise a `AttributeError`. [#2047]
- `astropy.units` now correctly preserves the precision of fractional powers. [#2070]
- If a `Unit` or `Quantity` is raised to a floating point power that is very close to a rational number with a denominator less than or equal to 10, it is converted to a `Fraction` object to preserve its precision through complex unit conversion operations. [#2070]

astropy.utils

- Fixed crash in `timer.RunTimePredictor.do_fit`. [#1905]
- Fixed `astropy.utils.compat.argparse` for Python 3.1. [#2017]

astropy.wcs

- `astropy.wcs.WCS`, `astropy.wcs.WCS.fix` and `astropy.wcs.find_all_wcs` now have a `translate_units` keyword argument that is passed down to `astropy.wcs.Wcsprm.fix`. This can be used to specify any unsafe translations of units from rarely used ones to more commonly used ones.

  Although `"S"` is commonly used to represent seconds, its translation to `"s"` is potentially unsafe since the standard recognizes `"S"` formally as Siemens, however rarely that may be used. The same applies to `"H"` for hours (Henry), and `"D"` for days (Debye).

  When these sorts of changes are performed, a warning is emitted. [#1854]

- When a unit is “fixed” by `astropy.wcs.WCS.fix` or `astropy.wcs.Wcsprm.unitfix`, it now correctly reports the `CUNIT` field that was changed. [#1854]

- `astropy.wcs.Wcs.printwcs` will no longer warn that `cdelt` is being ignored when none was present in the FITS file. [#1845]

- `astropy.wcs.Wcsprm.set` is called from within the `astropy.wcs.WCS` constructor, therefore any invalid information in the keywords will be raised from the constructor, rather than on a subsequent call to a transformation method. [#1918]

- Fix a memory corruption bug when using `astropy.wcs.Wcs.sub` with `astropy.wcs.WCSSUB_CELESTIAL`. [#1960]
- Fixed the `AttributeError` exception that was raised when using `astropy.wcs.WCS.footprint_to_file`. [#1912]
- Fixed a `NameError` exception that was raised when using `astropy.wcs.validate` or the `wcslint` script. [#2053]
- Fixed a bug where named WCSes may be erroneously reported as `' '` when using `astropy.wcs.validate` or the `wcslint` script. [#2053]
- Fixed a bug where error messages about incorrect header keywords may not be propagated correctly, resulting in a “NULL error object in wcslib” message. [#2106]

Misc

- There are a number of improvements to make Astropy work better on big endian platforms, such as MIPS, PPC, s390x and SPARC. [#1849]
- The test suite will now raise exceptions when a deprecated feature of Python or Numpy is used. [#1948]

Other Changes and Additions

- A new function, `astropy.wcs.get_include`, has been added to get the location of the `astropy.wcs` C header files. [#1755]
- The doctests in the `.rst` files in the `docs` folder are now tested along with the other unit tests. This is in addition to the testing of doctests in docstrings that was already being performed. See `docs/development/testguide.rst` for more information. [#1771]
- Fix a problem where import fails on Python 3 if setup.py exists in current directory. [#1877]

0.3 (2013-11-20)

New Features

- General
- A top-level configuration item, `unicode_output` has been added to control whether the Unicode string representation of certain objects will contain Unicode characters. For example, when `use_unicode` is `False` (default):

```
>>> from astropy import units as u
>>> print(unicode(u.degree))
deg
```

When `use_unicode` is `True`:

```
>>> from astropy import units as u
>>> print(unicode(u.degree))
°
```

See handling-unicode for more information. [#1441]

- `astropy.utils.misc.find_api_page` is now imported into the top-level. This allows usage like `astropy.find_api_page(astropy.units.Quantity)`. [#1779]

astropy.convolution

- New class-based system for generating kernels, replacing `make_kernel`. [#1255] The `astropy.nddata.convolution` sub-package has now been moved to `astropy.convolution`. [#1451]

astropy.coordinates

- Two classes `astropy.coordinates.Longitude` and `astropy.coordinates.Latitude` have been added. These are derived from the new `Angle` class and used for all longitude-like (RA, azimuth, galactic L) and latitude-like coordinates (Dec, elevation, galactic B) respectively. The `Longitude` class provides auto-wrapping capability and `Latitude` performs bounds checking.
- `astropy.coordinates.Distance` supports conversion to and from distance modulii. [#1472]
- `astropy.coordinates.SphericalCoordinateBase` and derived classes now support arrays of coordinates, enabling large speed-ups for some operations on multiple coordinates at the same time. These coordinates can also be indexed using standard slicing or any Numpy-compatible indexing. [#1535, #1615]
- Array coordinates can be matched to other array coordinates, finding the closest matches between the two sets of coordinates (see the `astropy.coordinates.matching.match_coordinates_3d` and `astropy.coordinates.matching.match_coordinates_sky` functions). [#1535]

astropy.cosmology

- Added support for including massive Neutrinos in the cosmology classes.

The Planck (2013) cosmology has been updated to use this. [#1364]
- Calculations now use and return `Quantity` objects where appropriate. [#1237]

astropy.io.ascii

- Added support for writing IPAC format tables [#1152].

astropy.io.fits

- Added initial support for table columns containing pseudo-unsigned integers. This is currently enabled by using the `uint=True` option when opening files; any table columns with the correct BZERO value will be interpreted and returned as arrays of unsigned integers. [#906]
- Upgraded vendored copy of CFITSIO to v3.35, though backwards compatibility back to version v3.28 is maintained.
- Added support for reading and writing tables using the Q format for columns. The Q format is identical to the P format (variable-length arrays) except that it uses 64-bit integers for the data descriptors, allowing more than 4 GB of variable-length array data in a single table.
- Some refactoring of the table and `FITS_rec` modules in order to better separate the details of the FITS binary and ASCII table data structures from the HDU data structures that encapsulate them. Most of these changes should not be apparent to users (but see API Changes below).

astropy.io.votable

- Updated to support the VOTable 1.3 draft. [#433]
- Added the ability to look up and group elements by their utype attribute. [#622]
- The format of the units of a VOTable file can be specified using the `unit_format` parameter. Note that units are still always written out using the CDS format, to ensure compatibility with the standard.

astropy.modeling

- Added a new framework for representing and evaluating mathematical models and for fitting data to models. See “What’s New in Astropy 0.3” in the documentation for further details. [#493]

astropy.stats

- Added robust statistics functions `astropy.stats.funcs.median_absolute_deviation`, `astropy.stats.funcs.biweight_location`, and `astropy.stats.funcs.biweight_midvariance`. [#621]

- Added `astropy.stats.funcs.signal_to_noise_oir_ccd` for computing the signal to noise ratio for source being observed in the optical/IR using a CCD. [#870]
- Add `axis=int` option to `stropy.stats.funcs.sigma_clip` to allow clipping along a given axis for multidimensional data. [#1083]

astropy.table

- New columns can be added to a table via assignment to a non-existing column by name. [#726]
- Added `join` function to perform a database-like join on two tables. This includes support for inner, left, right, and outer joins as well as metadata merging. [#903]
- Added `hstack` and `vstack` functions to stack two or more tables. [#937]
- Tables now have a `.copy` method and include support for `copy` and `deepcopy`. [#1208]
- Added support for selecting and manipulating groups within a table with a database style `group_by` method. [#1424]
- Table `read` and `write` functions now include rudimentary support reading and writing of FITS tables via the unified reading/writing interface. [#591]
- The `units` and `dtypes` attributes and keyword arguments in Column, MaskedColumn, Row, and Table are now deprecated in favor of the single-tense `unit` and `dtype`. [#1174]
- Setting a column from a Quantity now correctly sets the unit on the Column object. [#732]
- Add `remove_row` and `remove_rows` to remove table rows. [#1230]
- Added a new `Table.show_in_browser` method that opens a web browser and displays the table rendered as HTML. [#1342]
- New tables can now be instantiated using a single row from an existing table. [#1417]

astropy.time

- New `Time` objects can be instantiated from existing `Time` objects (but with different format, scale, etc.) [#889]
- Added a `Time.now` classmethod that returns the current UTC time, similarly to Python's `datetime.now`. [#1061]
- Update internal time manipulations so that arithmetic with Time and TimeDelta objects maintains sub-nanosecond precision over a time span longer than the age of the universe. [#1189]
- Use `astropy.utils.iers` to provide `delta_ut1_utc`, so that automatic calculation of UT1 becomes possible. [#1145]

- Add `datetime` format which allows converting to and from standard library `datetime.datetime` objects. [#860]
- Add `plot_date` format which allows converting to and from the date representation used when plotting dates with matplotlib via the `matplotlib.pyplot.plot_date` function. [#860]
- Add `gps` format (seconds since 1980-01-01 00:00:00 UTC, including leap seconds) [#1164]
- Add array indexing to Time objects [#1132]
- Allow for arithmetic of multi-element and single-element Time and TimeDelta objects. [#1081]
- Allow multiplication and division of TimeDelta objects by constants and arrays, as well as changing sign (negation) and taking the absolute value of TimeDelta objects. [#1082]
- Allow comparisons of Time and TimeDelta objects. [#1171]
- Support interaction of Time and Quantity objects that represent a time interval. [#1431]

astropy.units

- Added parallax equivalency for length-angle. [#985]
- Added mass-energy equivalency. [#1333]
- Added a new-style format method which will use format specifiers (like `0.03f`) in new-style format strings for the Quantity's value. Specifiers which can't be applied to the value will fall back to the entire string representation of the quantity. [#1383]
- Added support for complex number values in quantities. [#1384]
- Added new spectroscopic equivalencies for velocity conversions (relativistic, optical, and radio conventions are supported) [#1200]
- The `spectral` equivalency now also handles wave number.
- The `spectral_density` equivalency now also accepts a Quantity for the frequency or wavelength. It also handles additional flux units.
- Added Brightness Temperature (antenna gain) equivalency for conversion between \(T_B\) and flux density. [#1327]
- Added percent unit, and allowed any string containing just a number to be interpreted as a scaled dimensionless unit. [#1409]
- New-style format strings can be used to set the unit output format. For example, `"{0:latex}".format(u.km)` will print with the latex formatter. [#1462]
- The `Unit.is_equivalent` method can now take a tuple. In this case, the method returns `True` if the unit is equivalent to any of the units listed in the tuple. [#1521]
- `def_unit` can now take a 2-tuple of names of the form (short, long), where each entry is a list. This allows for handling strange units that might

have multiple short names. [#1543]
- Added `dimensionless_angles` equivalency, which allows conversion of any power of radian to dimensionless. [#1161]
- Added the ability to enable set of units, or equivalencies that are used by default. Also provided context managers for these cases. [#1268]
- Imperial units are disabled by default. [#1593, #1662]
- Added an `astropy.units.add_enabled_units` context manager, which allows creating a temporary context with additional units temporarily enabled in the global units namespace. [#1662]
- `Unit` instances now have `.si` and `.cgs` properties a la `Quantity`. These serve as shortcuts for `Unit.to_system(cgs)[0]` etc. [#1610]

astropy.vo

- New package added to support Virtual Observatory Simple Cone Search query and service validation. [#552]

astropy.wcs

- Fixed attribute error in `astropy.wcs.Wcsprm` (lattype->lattyp) [#1463]
- Included a new command-line script called `wcslint` and accompanying API for validating the WCS in a given FITS file or header. [#580]
- Upgraded included version of WCSLIB to 4.19.

astropy.utils

- Added a new set of utilities in `astropy.utils.timer` for analyzing the runtime of functions and making runtime predections for larger inputs. [#743]
- `ProgressBar` and `Spinner` classes can now be used directly to return generator expressions. [#771]
- Added `astropy.utils.iers` which allows reading in of IERS A or IERS B bulletins and interpolation in UT1-UTC.
- Added a function `astropy.utils.find_api_page`–given a class or object from the `astropy` package, this will open that class's API documentation in a web browser. [#663]
- Data download functions such as `download_file` now accept a `show_progress` argument to suppress console output, and a `timeout` argument. [#865, #1258]

astropy.extern.six

- Added six for python2/python3 compatibility
- Astropy now uses the ERFA library instead of the IAU SOFA library for fundamental time transformation routines. The ERFA library is derived, with

permission, from the IAU SOFA library but is distributed under a BSD license. See `license/ERFA.rst` for details. [#1293]

astropy.logger

- The Astropy logger now no longer catches exceptions by default, and also only captures warnings emitted by Astropy itself (prior to this change, following an import of Astropy, any warning got re-directed through the Astropy logger). Logging to the Astropy log file has also been disabled by default. However, users of Astropy 0.2 will likely still see the previous behavior with Astropy 0.3 for exceptions and logging to file since the default configuration file installed by 0.2 set the exception logging to be on by default. To get the new behavior, set the `log_exceptions` and `log_to_file` configuration items to `False` in the `astropy.cfg` file. [#1331]

API Changes

- General
- The configuration option `utils.console.use_unicode` has been moved to the top level and renamed to `unicode_output`. It now not only affects console widgets, such as progress bars, but also controls whether calling **`unicode`** on certain classes will return a string containing unicode characters.

astropy.coordinates

- The `astropy.coordinates.Angle` class is now a subclass of `astropy.units.Quantity`. This means it has all of the methods of a **`numpy.ndarray`**. [#1006]
- The `astropy.coordinates.Distance` class is now a subclass of `astropy.units.Quantity`. This means it has all of the methods of a **`numpy.ndarray`**. [#1472]
- All angular units are now supported, not just `radian`, `degree` and `hour`, but now `arcsecond` and `arcminute` as well. The object will retain its native unit, so when printing out a value initially provided in hours, its `to_string()` will, by default, also be expressed in hours.
- The `Angle` class now supports arrays of angles.
- To be consistent with `units.Unit`, `Angle.format` has been deprecated and renamed to `Angle.to_string`.
- To be consistent with `astropy.units`, all plural forms of unit names have been removed. Therefore, the following properties of `astropy.coordinates.Angle` should be renamed:

- `radians` -> `radian`
- `degrees` -> `degree`
- `hours` -> `hour`
- Multiplication and division of two `Angle` objects used to raise `NotImplementedError`. Now they raise `TypeError`.
- The `astropy.coordinates.Angle` class no longer has a `bounds` attribute so there is no bounds-checking or auto-wrapping at this level. This allows `Angle` objects to be used in arbitrary arithmetic expressions (e.g. coordinate distance computation).
- The `astropy.coordinates.RA` and `astropy.coordinates.Dec` classes have been removed and replaced with `astropy.coordinates.Longitude` and `astropy.coordinates.Latitude` respectively. These are now used for the components of Galactic and Horizontal (Alt-Az) coordinates as well instead of plain `Angle` objects.
- `astropy.coordinates.angles.rotation_matrix` and `astropy.coordinates.angles.angle_axis` now take a `unit` kwarg instead of `degrees` kwarg to specify the units of the angles. `rotation_matrix` will also take the unit from the given `Angle` object if no unit is provided.
- The `AngularSeparation` class has been removed. The output of the coordinates `separation()` method is now an `astropy.coordinates.Angle`. [#1007]
- The coordinate classes have been renamed in a way that remove the `Coordinates` at the end of the class names. E.g., `ICRSCoordinates` from previous versions is now called `ICRS`. [#1614]
- `HorizontalCoordinates` are now named `AltAz`, to reflect more common terminology.

astropy.cosmology

- The Planck (2013) cosmology will likely give slightly different (and more accurate) results due to the inclusion of Neutrino masses. [#1364]
- Cosmology class properties now return `Quantity` objects instead of simple floating-point values. [#1237]
- The names of cosmology instances are now truly optional, and are set to `None` rather than the name of the class if the user does not provide them. [#1705]

astropy.io.ascii

- In the `read` method of `astropy.io.ascii`, empty column values in an

ASCII table are now treated as missing values instead of the previous treatment as a zero-length string “”. This now corresponds to the behavior of other table readers like `numpy.genfromtxt`. To restore the previous behavior set `fill_values=None` in the call to `ascii.read()`. [#919]
- The `read` and `write` methods of `astropy.io.ascii` now have a `format` argument for specifying the file format. This is the preferred way to choose the format instead of the `Reader` and `Writer` arguments. [#961]
- The `include_names` and `exclude_names` arguments were removed from the `BaseHeader` initializer, and now instead handled by the reader and writer classes directly. [#1350]
- Allow numeric and otherwise unusual column names when reading a table where the `format` argument is specified, but other format details such as the delimiter or quote character are being guessed. [#1692]
- When reading an ASCII table using the `Table.read()` method, the default has changed from `guess=False` to `guess=True` to allow auto-detection of file format. This matches the default behavior of `ascii.read()`.

astropy.io.fits

- The `astropy.io.fits.new_table` function is marked “pending deprecation”. This does not mean it will be removed outright or that its functionality has changed. It will likely be replaced in the future for a function with similar, if not subtly different functionality. A better, if not slightly more verbose approach is to use `pyfits.FITS_rec.from_columns` to create a new `FITS_rec` table–this has the same interface as `pyfits.new_table`. The difference is that it returns a plan `FITS_rec` array, and not an HDU instance. This `FITS_rec` object can then be used as the data argument in the constructors for `BinTableHDU` (for binary tables) or `TableHDU` (for ASCII tables). This is analogous to creating an `ImageHDU` by passing in an image array. `pyfits.FITS_rec.from_columns` is just a simpler way of creating a FITS-compatible recarray from a FITS column specification.
- The `updateHeader`, `updateHeaderData`, and `updateCompressedData` methods of the `CompDataHDU` class are pending deprecation and moved to internal methods. The operation of these methods depended too much on internal state to be used safely by users; instead they are invoked automatically in the appropriate places when reading/writing compressed image HDUs.
- The `CompDataHDU.compData` attribute is pending deprecation in favor of the clearer and more PEP-8 compatible `CompDataHDU.compressed_data`.

- The constructor for `CompDataHDU` has been changed to accept new keyword arguments. The new keyword arguments are essentially the same, but are in underscore_separated format rather than camelCase format. The old arguments are still pending deprecation.
- The internal attributes of HDU classes `_hdrLoc`, `_datLoc`, and `_datSpan` have been replaced with `_header_offset`, `_data_offset`, and `_data_size` respectively. The old attribute names are still pending deprecation. This should only be of interest to advanced users who have created their own HDU subclasses.
- The following previously deprecated functions and methods have been removed entirely: `createCard`, `createCardFromString`, `upperKey`, `ColDefs.data`, `setExtensionNameCaseSensitive`, `_File.getfile`, `_TableBaseHDU.get_coldefs`, `Header.has_key`, `Header.ascardlist`.
- Interfaces that were pending deprecation are now fully deprecated. These include: `create_card`, `create_card_from_string`, `upper_key`, `Header.get_history`, and `Header.get_comment`.
- The `.name` attribute on HDUs is now directly tied to the HDU's header, so that if `.header['EXTNAME']` changes so does `.name` and vice-versa.

astropy.io.registry

- Identifier functions for reading/writing Table and NDData objects should now accept `(origin, *args, **kwargs)` instead of `(origin, args, kwargs)`. [#591]
- Added a new `astropy.io.registry.get_formats` function for listing registered I/O formats and details about the their readers/writers. [#1669]

astropy.io.votable

- Added a new option `use_names_over_ids` option to use when converting from VOTable objects to Astropy Tables. This can prevent a situation where column names are not preserved when converting from a VOTable. [#609]

astropy.nddata

- The `astropy.nddata.convolution` sub-package has now been moved to `astropy.convolution`, and the `make_kernel` function has been removed. (the kernel classes should be used instead) [#1451]

astropy.stats.funcs

- For `sigma_clip`, the `maout` optional parameter has been removed, and the function now always returns a masked array. A new boolean parameter `copy` can be used to indicated whether the input data should be copied (`copy=True`, default) or used by reference (`copy=False`) in the output masked array. [#1083]

astropy.table

- The first argument to the `Column` and `MaskedColumn` classes is now the data array–the `name` argument has been changed to an optional keyword argument. [#840]
- Added support for instantiating a `Table` from a list of dict, each one representing a single row with the keys mapping to column names. [#901]
- The plural 'units' and 'dtypes' have been switched to 'unit' and 'dtype' where appropriate. The original attributes are still present in this version as deprecated attributes, but will be removed in the next version. [#1174]
- The `copy` methods of `Column` and `MaskedColumn` were changed so that the first argument is now `order='C'`. This is required for compatibility with Numpy 1.8 which is currently in development. [#1250]
- Comparing a column (with == or !=) to a scalar, an array, or another column now always returns a boolean Numpy array (which is a masked array if either of the arguments in the comparison was masked). This is in contrast to the previous behavior, which in some cases returned a boolean Numpy array, and in some cases returned a boolean Column object. [#1446]

astropy.time

- For consistency with `Quantity`, the attributes `val` and `is_scalar` have been renamed to `value` and `isscalar`, respectively, and the attribute `vals` has been dropped. [#767]
- The double-float64 internal representation of time is used more efficiently to enable better accuracy. [#366]
- Format and scale arguments are now allowed to be case-insensitive. [#1128]

astropy.units

- The `Quantity` class now inherits from the Numpy array class, and includes the following API changes [#929]:
- Using `float(...)`, `int(...)`, and `long(...)` on a quantity will now only work if the quantity is dimensionless and unscaled.
- All Numpy ufuncs should now treat units correctly (or raise an exception if not supported), rather than extract the value of quantities and operate on this, emitting a warning about the implicit loss of units.
- When using relevant Numpy ufuncs on dimensionless quantities (e.g. `np.exp(h * nu / (k_B * T))`), or combining dimensionless quantities with Python scalars or plain Numpy arrays `1 + v / c`, the dimensionless

Quantity will automatically be converted to an unscaled dimensionless Quantity.
- When initializing a quantity from a value with no unit, it is now set to be dimensionless and unscaled by default. When initializing a Quantity from another Quantity and with no unit specified in the initializer, the unit is now taken from the unit of the Quantity being initialized from.
- Strings are no longer allowed as the values for Quantities. [#1005]
- Quantities are always comparable with zero regardless of their units. [#1254]
- The exception `astropy.units.UnitsException` has been renamed to `astropy.units.UnitsError` to be more consistent with the naming of built-in Python exceptions. [#1406]
- Multiplication with and division by a string now always returns a Unit (rather than a Quantity when the string was first) [#1408]
- Imperial units are disabled by default.

astropy.wcs

- For those including the `astropy.wcs` C headers in their project, they should now include it as:

  #include “astropy_wcs/astropy_wcs_api.h”

  instead of:

  #include “astropy_wcs_api.h”

  [#1631]
- The `--enable-legacy` option for `setup.py` has been removed. [#1493]

Bug Fixes

astropy.io.ascii

- The `write()` function was ignoring the `fill_values` argument. [#910]
- Fixed an issue in `DefaultSplitter.join` where the delimiter attribute was ignored when writing the CSV. [#1020]
- Fixed writing of IPAC tables containing null values. [#1366]
- When a table with no header row was read without specifying the format and using the `names` argument, then the first row could be dropped. [#1692]

astropy.io.fits

- Binary tables containing compressed images may, optionally, contain other columns unrelated to the tile compression convention. Although this is an uncommon use case, it is permitted by the standard.
- Reworked some of the file I/O routines to allow simpler, more consistent

mapping between OS-level file modes ('rb', 'wb', 'ab', etc.) and the more "PyFITS-specific" modes used by PyFITS like "readonly" and "update". That is, if reading a FITS file from an open file object, it doesn't matter as much what "mode" it was opened in so long as it has the right capabilities (read/write/etc.) Also works around bugs in the Python io module in 2.6+ with regard to file modes.
- Fixed a long-standing issue where writing binary tables did not correctly write the TFORMn keywords for variable-length array columns (they omitted the max array length parameter of the format). This was thought fixed in an earlier version, but it was only fixed for compressed image HDUs and not for binary tables in general.

astropy.nddata

- Fixed crash when trying to multiple or divide `NDData` objects with uncertainties. [#1547]

astropy.table

- Using a list of strings to index a table now correctly returns a new table with the columns named in the list. [#1454]
- Inequality operators now work properly with `Column` objects. [#1685]

astropy.time

- `Time` scale and format attributes are now shown when calling `dir()` on a `Time` object. [#1130]

astropy.wcs

- Fixed assignment to string-like WCS attributes on Python 3. [#956]

astropy.units

- Fixed a bug that caused the order of multiplication/division of plain Numpy arrays with Quantities to matter (i.e. if the plain array comes first the units were not preserved in the output). [#899]
- Directly instantiated `CompositeUnits` were made printable without crashing. [#1576]

Misc

- Fixed various modules that hard-coded `sys.stdout` as default arguments to functions at import time, rather than using the runtime value of `sys.stdout`. [#1648]
- Minor documentation fixes and enhancements [#922, #1034, #1210, #1217,

#1491, #1492, #1498, #1582, #1608, #1621, #1646, #1670, #1756]

- Fixed a crash that could sometimes occur when running the test suite on systems with platform names containing non-ASCII characters. [#1698]

Other Changes and Additions

- General
- Astropy now follows the PSF Code of Conduct. [#1216]
- Astropy's test suite now tests all doctests in inline docstrings. Support for running doctests in the reST documentation is planned to follow in v0.3.1.
- Astropy's test suite can be run on multiple CPUs in parallel, often greatly improving runtime, using the `--parallel` option. [#1040]
- A warning is now issued when using Astropy with Numpy < 1.5–much of Astropy may still work in this case but it shouldn't be expected to either. [#1479]
- Added automatic download/build/installation of Numpy during Astropy installation if not already found. [#1483]
- Handling of metadata for the `NDData` and `Table` classes has been unified by way of a common `MetaData` descriptor–it allows instantiating an object with metadata of any mapping type, and subsequently prevents replacing the mapping stored in the `.meta` attribute (only direct updates to that object are allowed). [#1686]

astropy.coordinates

- Angles containing out of bounds minutes or seconds (e.g. 60) can be parsed–the value modulo 60 is used with carry to the hours/minutes, and a warning is issued rather than raising an exception. [#990]

astropy.io.fits

- The new compression code also adds support for the ZQUANTIZ and ZDITHER0 keywords added in more recent versions of this FITS Tile Compression spec. This includes support for lossless compression with GZIP. (#198) By default no dithering is used, but the `SUBTRACTIVE_DITHER_1` and `SUBTRACTIVE_DITHER_2` methods can be enabled by passing the correct constants to the `quantize_method` argument to the `CompImageHDU` constructor. A seed can be manually specified, or automatically generated using either the system clock or checksum-based methods via the `dither_seed` argument. See the documentation for `CompImageHDU` for more details.
- Images compressed with the Tile Compression standard can now be larger than 4 GB through support of the Q format.
- All HDUs now have a `.ver` `.level` attribute that returns the value of the

EXTVAL and EXTLEVEL keywords from that HDU's header, if the exist. This was added for consistency with the `.name` attribute which returns the EXTNAME value from the header.
- Then `Column` and `ColDefs` classes have new `.dtype` attributes which give the Numpy dtype for the column data in the first case, and the full Numpy compound dtype for each table row in the latter case.
- There was an issue where new tables created defaulted the values in all string columns to '0.0'. Now string columns are filled with empty strings by default–this seems a less surprising default, but it may cause differences with tables created with older versions of PyFITS or Astropy.

astropy.io.misc

- The HDF5 reader can now refer to groups in the path as well as datasets; if given a group, the first dataset in that group is read. [#1159]

astropy.nddata

- `NDData` objects have more helpful, though still rudimentary `__str__` and ``__repr__`` displays. [#1313]

astropy.units

- Added 'cycle' unit. [#1160]
- Extended units supported by the CDS formatter/parser. [#1468]
- Added unicode an LaTeX symbols for liter. [#1618]

astropy.wcs

- Redundant SCAMP distortion parameters are removed with SIP distortions are also present. [#1278]
- Added iterative implementation of `all_world2pix` that can be reliably inverted. [#1281]

0.2.5 (2013-10-25)

Bug Fixes

astropy.coordinates

- Fixed incorrect string formatting of Angles using `precision=0`. [#1319]
- Fixed string formatting of Angles using `decimal=True` which ignored the `precision` argument. [#1323]
- Fixed parsing of format strings using appropriate unicode characters instead of the ASCII `-` for minus signs. [#1429]

astropy.io.ascii

- Fixed a crash in the IPAC table reader when the `include/exclude_names` option is set. [#1348]
- Fixed writing AASTex tables to honor the `tabletype` option. [#1372]

astropy.io.fits

- Improved round-tripping and preservation of manually assigned column attributes (`TNULLn`, `TSCALn`, etc.) in table HDU headers. (Note: This issue was previously reported as fixed in Astropy v0.2.2 by mistake; it is not fixed until v0.3.) [#996]
- Fixed a bug that could cause a segfault when trying to decompress an compressed HDU whose contents are truncated (due to a corrupt file, for example). This still causes a Python traceback but better that than a segfault. [#1332]
- Newly created `CompImageHDU` HDUs use the correct value of the `DEFAULT_COMPRESSION_TYPE` module-level constant instead of hard-coding "RICE_1" in the header.
- Fixed a corner case where when extra memory is allocated to compress an image, it could lead to unnecessary in-memory copying of the compressed image data and a possible memory leak through Numpy.
- Fixed a bug where assigning from an mmap'd array in one FITS file over the old (also mmap'd) array in another FITS file failed to update the destination file. Corresponds to PyFITS issue 25.
- Some miscellaneous documentation fixes.

astropy.io.votable

- Added a warning for when a VOTable 1.2 file contains no `RESOURCES` elements (at least one should be present). [#1337]
- Fixed a test failure specific to MIPS architecture caused by an errant floating point warning. [#1179]

astropy.nddata.convolution

- Prevented in-place modification of the input arrays to `convolve()`. [#1153]

astropy.table

- Added HTML escaping for string values in tables when outputting the table as HTML. [#1347]
- Added a workaround in a bug in Numpy that could cause a crash when accessing a table row in a masked table containing `dtype=object` columns. [#1229]

- Fixed an issue similar to the one in #1229, but specific to unmasked tables. [#1403]

astropy.units

- Improved error handling for unparseable units and fixed parsing CDS units without mantissas in the exponent. [#1288]
- Added a physical type for spectral flux density. [#1410]
- Normalized conversions that should result in a scale of exactly 1.0 to round off slight floating point imprecisions. [#1407]
- Added support in the CDS unit parser/formatter for unusual unit prefixes that are nonetheless required to be supported by that convention. [#1426]
- Fixed the parsing of `sqrt()` in unit format strings which was returning `unit ** 2` instead of `unit ** 0.5`. [#1458]

astropy.wcs

- When passing a single array to the wcs transformation functions, (`astropy.wcs.Wcs.all_pix2world`, etc.), its second dimension must now exactly match the number of dimensions in the transformation. [#1395]
- Improved error message when incorrect arguments are passed to `WCS.wcs_world2pix`. [#1394]
- Fixed a crash when trying to read WCS from FITS headers on Python 3.3 in Windows. [#1363]
- Only headers that are required as part of the WCSLIB C API are installed by the package, per request of system packagers. [#1666]

Misc

- Fixed crash when the `COLUMNS` environment variable is set to a non-integer value. [#1291]
- Fixed a bug in `ProgressBar.map` where `multiprocess=True` could cause it to hang on waiting for the process pool to be destroyed. [#1381]
- Fixed a crash on Python 3.2 when affiliated packages try to use the `astropy.utils.data.get_pkg_data_*` functions. [#1256]
- Fixed a minor path normalization issue that could occur on Windows in `astropy.utils.data.get_pkg_data_filename`. [#1444]
- Fixed an annoyance where configuration items intended only for testing showed up in users' astropy.cfg files. [#1477]
- Prevented crashes in exception logging in unusual cases where no traceback is associated with the exception. [#1518]
- Fixed a crash when running the tests in unusual environments where `sys.stdout.encoding` is `None`. [#1530]
- Miscellaneous documentation fixes and improvements [#1308, #1317,

#1377, #1393, #1362, #1516]

Other Changes and Additions

- Astropy installation now requests setuptools >= 0.7 during build/installation if neither distribute or setuptools >= 0.7 is already installed. In other words, if `import setuptools` fails, `ez_setup.py` is used to bootstrap the latest setuptools (rather than using `distribute_setup.py` to bootstrap the now obsolete distribute package). [#1197]
- When importing Astropy from a source checkout without having built the extension modules first an `ImportError` is raised rather than a `SystemExit` exception. [#1269]

0.2.4 (2013-07-24)

Bug Fixes

astropy.coordinates

- Fixed the angle parser to support parsing the string “1 degree”. [#1168]

astropy.cosmology

- Fixed a crash in the `comoving_volume` method on non-flat cosmologies when passing it an array of redshifts.

astropy.io.ascii

- Fixed a bug that prevented saving changes to the comment symbol when writing changes to a table. [#1167]

astropy.io.fits

- Added a workaround for a bug in 64-bit OSX that could cause truncation when writing files greater than 2^32 bytes in size. [#839]

astropy.io.votable

- Fixed incorrect reading of tables containing multiple `<RESOURCE>` elements. [#1223]

astropy.table

- Fixed a bug where `Table.remove_column` and `Table.rename_column` could cause a masked table to lose its masking. [#1120]

- Fixed bugs where subclasses of `Table` did not preserver their class in certain operations. [#1142]
- Fixed a bug where slicing a masked table did not preserve the mask. [#1187]

astropy.units

- Fixed a bug where the `.si` and `.cgs` properties of dimensionless `Quantity` objects raised a `ZeroDivisionError`. [#1150]
- Fixed a bug where multiple subsequent calls to the `.decompose()` method on array quantities applied a scale factor each time. [#1163]

Misc

- Fixed an installation crash that could occur sometimes on Debian/Ubuntu and other *NIX systems where `pkg_resources` can be installed without installing `setuptools`. [#1150]
- Updated the `distribute_setup.py` bootstrapper to use setuptools >= 0.7 when installing on systems that don't already have an up to date version of distribute/setuptools. [#1180]
- Changed the `version.py` template so that Astropy affiliated packages can (and they should) use their own `cython_version.py` and `utils._compiler` modules where appropriate. This issue only pertains to affiliated package maintainers. [#1198]
- Fixed a corner case where the default config file generation could crash if building with matplotlib but *not* Sphinx installed in a virtualenv. [#1225]
- Fixed a crash that could occur in the logging module on systems that don't have a default preferred encoding (in particular this happened in some versions of PyCharm). [#1244]
- The Astropy log now supports passing non-string objects (and calling `str()` on them by default) to the logging methods, in line with Python's standard logging API. [#1267]
- Minor documentation fixes [#582, #696, #1154, #1194, #1212, #1213, #1246, #1252]

Other Changes and Additions

astropy.cosmology

- Added a new `Plank13` object representing the Plank 2013 results. [#895]

astropy.units

- Performance improvements in initialization of `Quantity` objects with a large number of elements. [#1231]

0.2.3 (2013-05-30)

Bug Fixes

astropy.time

- Fixed inaccurate handling of leap seconds when converting from UTC to UNIX timestamps. [#1118]
- Tightened required accuracy in many of the time conversion tests. [#1121]

Misc

- Fixed a regression that was introduced in v0.2.2 by the fix to issue #992 that was preventing installation of Astropy affiliated packages that use Astropy's setup framework. [#1124]

0.2.2 (2013-05-21)

Bug Fixes

astropy.io

- Fixed issues in both the `fits` and `votable` sub-packages where array byte order was not being handled consistently, leading to possible crashes especially on big-endian systems. [#1003]

astropy.io.fits

- When an error occurs opening a file in fitsdiff the exception message will now at least mention which file had the error.
- Fixed a couple cases where creating a new table using TDIMn in some of the columns could cause a crash.
- Slightly refactored how tables containing variable-length array columns are handled to add two improvements: Fixes an issue where accessing the data after a call to the `astropy.io.fits.getdata` convenience function caused an exception, and allows the VLA data to be read from an existing mmap of the FITS file.
- Fixed a bug on Python 3 where attempting to open a non-existent file on Python 3 caused a seemingly unrelated traceback.
- Fixed an issue in the tests that caused some tests to fail if Astropy is installed with read-only permissions.
- Fixed a bug where instantiating a `BinTableHDU` from a numpy array containing boolean fields converted all the values to `False`.
- Fixed an issue where passing an array of integers into the constructor of

`Column()` when the column type is floats of the same byte width caused the column array to become garbled.
- Fixed inconsistent behavior in creating CONTINUE cards from byte strings versus unicode strings in Python 2–CONTINUE cards can now be created properly from unicode strings (so long as they are convertable to ASCII).
- Fixed a bug in parsing HIERARCH keywords that do not have a space after the first equals sign (before the value).
- Prevented extra leading whitespace on HIERARCH keywords from being treated as part of the keyword.
- Fixed a bug where HIERARCH keywords containing lower-case letters was mistakenly marked as invalid during header validation along with an ancillary issue where the `Header.index()` method id not work correctly with HIERARCH keywords containing lower-case letters.
- Disallowed assigning NaN and Inf floating point values as header values, since the FITS standard does not define a way to represent them in. Because this is undefined, the previous behavior did not make sense and produced invalid FITS files. [#954]
- Fixed an obscure issue that can occur on systems that don’t have flush to memory-mapped files implemented (namely GNU Hurd). [#968]

astropy.io.votable

- Stopped deprecation warnings from the `astropy.io.votable` package that could occur during setup. [#970]
- Fixed an issue where INFO elements were being incorrectly dropped when occurring inside a TABLE element. [#1000]
- Fixed obscure test failures on MIPS platforms. [#1010]

astropy.nddata.convolution

- Fixed an issue in `make_kernel()` when using an Airy function kernel. Also removed the superfluous ‘brickwall’ option. [#939]

astropy.table

- Fixed a crash that could occur when adding a row to an empty (rowless) table with masked columns. [#973]
- Made it possible to assign to one table row from the value of another row, effectively making it easier to copy rows, for example. [#1019]

astropy.time

- Added appropriate `__copy__` and `__deepcopy__` behavior; this omission caused a seemingly unrelated error in FK5 coordinate separation. [#891]

astropy.units

- Fixed an issue where the `isiterable()` utility returned `True` for quantities with scalar values. Added an `__iter__` method for the `Quantity` class and fixed `isiterable()` to catch false positives. [#878]
- Fixed previously undefined behavior when multiplying a unit by a string. [#949]
- Added 'time' as a physical type–this was a simple omission. [#959]
- Fixed issues with pickling unit objects so as to play nicer with the multiprocessing module. [#974]
- Made it more difficult to accidentally override existing units with a new unit of the same name. [#1070]
- Added several more physical types and units that were previously omitted, including 'mass density', 'specific volume', 'molar volume', 'momentum', 'angular momentum', 'angular speed', 'angular acceleration', 'electric current', 'electric current density', 'electric field strength', 'electric flux density', 'electric charge density', 'permittivity', 'electromagnetic field strength', 'radiant intensity', 'data quantity', 'bandwidth'; and 'knots', 'nautical miles', 'becquerels', and 'curies' respectively. [#1072]

Misc

- Fixed a permission error that could occur when running `astropy.test()` on Python 3 when Astropy is installed as root. [#811]
- Made it easier to filter warnings from the `convolve()` function and from `Quantity` objects. [#853]
- Fixed a crash that could occur in Python 3 when generation of the default config file fails during setup. [#952]
- Fixed an unrelated error message that could occur when trying to import astropy from a source checkout without having build the extension modules first. This issue was claimed to be fixed in v0.2.1, but the fix itself had a bug. [#971]
- Fixed a crash that could occur when running the `build_sphinx` setup command in Python 3. [#977]
- Added a more helpful error message when trying to run the `setup.py build_sphinx` command when Sphinx is not installed. [#1027]
- Minor documentation fixes and restructuring. [#935, #967, #978, #1004, #1028, #1047]

Other Changes and Additions

- Some performance improvements to the `astropy.units` package, in particular improving the time it takes to import the sub-package. [#1015]

0.2.1 (2013-04-03)

Bug Fixes

astropy.coordinates

- Fixed encoding errors that could occur when formatting coordinate objects in code using `from __future__ import unicode_literals`. [#817]
- Fixed a bug where the minus sign was dropped when string formatting dms coordinates with -0 degrees. [#875]

astropy.io.fits

- Properly supports the ZQUANTIZ keyword used to support quantization level–this includes working support for lossless GZIP compression of images.
- Fixed support for opening gzipped FITS files in a writeable mode. [#256]
- Added a more helpful exception message when trying to read invalid values from a table when the required `TNULLn` keyword is missing. [#309]
- More refactoring of the tile compression handling to work around a potential memory access violation that was particularly prevalent on Windows. [#507]
- Fixed an integer size mismatch in the compression module that could affect 32-bit systems. [#786]
- Fixed malformatting of the `TFORMn` keywords when writing compressed image tables (they omitted the max array length parameter from the variable-length array format).
- Fixed a crash that could occur when writing a table containing multi-dimensional array columns from an existing file into a new file.
- Fixed a bug in fitsdiff that reported two header keywords containing NaN as having different values.

astropy.io.votable

- Fixed links to the `astropy.io.votable` documentation in the VOTable validator output. [#806]
- When reading VOTables containing integers that are out of range for their column type, display a warning rather than raising an exception. [#825]
- Changed the default string format for floating point values for better round-tripping. [#856]
- Fixed opening VOTables through the `Table.read()` interface for tables that have no names. [#927]
- Fixed creation of VOTables from an Astropy table that does not have a data mask. [#928]
- Minor documentation fixes. [#932]

astropy.nddata.convolution

- Added better handling of `inf` values to the `convolve_fft` family of functions. [#893]

astropy.table

- Fixed silent failure to assign values to a row on multiple columns. [#764]
- Fixed various buggy behavior when viewing a table after sorting by one of its columns. [#829]
- Fixed using `numpy.where()` with table indexing. [#838]
- Fixed a bug where opening a remote table with `Table.read()` could cause the entire table to be downloaded twice. [#845]
- Fixed a bug where `MaskedColumn` no longer worked if the column being masked is renamed. [#916]

astropy.units

- Added missing capability for array `Quantity`s to be initializable by a list of `Quantity`s. [#835]
- Fixed the definition of year and lightyear to be in terms of Julian year per the IAU definition. [#861]
- “degree” was removed from the list of SI base units. [#863]

astropy.wcs

- Fixed `TypeError` when calling `WCS.to_header_string()`. [#822]
- Added new method `WCS.all_world2pix` for converting from world coordinates to pixel space, including inversion of the astrometric distortion correction. [#1066, #1281]

Misc

- Fixed a minor issue when installing with `./setup.py develop` on a fresh git clone. This is likely only of interest to developers on Astropy. [#725]
- Fixes a crash with `ImportError: No module named 'astropy.version'` when running setup.py from a source checkout for the first time on OSX with Python 3.3. [#820]
- Fixed an installation issue where running `./setup.py install` or when installing with pip the `.astropy` directory gets created in the home directory of the user running the command. The user’s `.astropy` directory should only be created when they use Astropy, not when they install it. [#867]
- Fixed an exception when creating a `ProgressBar` with a “total” of 0. [#752]

- Added better documentation of behavior that can occur when trying to import the astropy package from within a source checkout without first building the extension modules. [#795, #864]
- Added link to the installation instructions in the README. [#797]
- Catches segfaults in xmllint which can occur sometimes and is otherwise out of our control. [#803]
- Minor changes to the documentation template. [#805]
- Fixed a minor exception handling bug in `download_file()`. [#808]
- Added cleanup of any temporary files if an error occurs in `download_file()`. [#857]
- Filesystem free space is checked for before attempting to download a file with `download_file()`. [#858]
- Fixed package data locating to work across symlinks–required to work with some OS packaging layouts. [#827]
- Fixed a bug when building Cython extensions where hidden files containing `.pyx` extensions could cause the build to crash. This can be an issue with software and filesystems that autogenerate hidden files. [#834]
- Fixed bug that could cause a “script” called README.rst to be installed in a bin directory. [#852]
- Fixed some miscellaneous and mostly rare reference leaks caught by cpychecker. [#914]

Other Changes and Additions

- Added logo and branding for Windows binary installers. [#741]
- Upgraded included version libexpat to 2.1.0. [#781]
- ~25% performance improvement in unit composition/decomposition. [#836]
- Added previously missing LaTeX formatting for `L_sun` and `R_sun`. [#841]
- ConfigurationItems now have a more useful and informative __repr__ and improved documentation for how to use them. [#855]
- Added a friendlier error message when trying to import astropy from a source checkout without first building the extension modules inplace. [#864]
- py.test now outputs more system information for help in debugging issues from users. [#869]
- Added unit definitions “mas” and “uas” for “milliarcsecond” and “microarcsecond” respectively. [#892]

0.2 (2013-02-19)

New Features

astropy.coordinates

- This new subpackage contains a representation of celestial coordinates, and provides a wide range of related functionality. While fully-functional, it is a work in progress and parts of the API may change in subsequent releases.

astropy.cosmology

- Update to include cosmologies with variable dark energy equations of state. (This introduces some API incompatibilities with the older Cosmology objects).
- Added parameters for relativistic species (photons, neutrinos) to the astropy.cosmology classes. The current treatment assumes that neutrinos are massless. [#365]
- Add a WMAP9 object using the final (9-year) WMAP parameters from Hinshaw et al. 2013. It has also been made the default cosmology. [#629, #724]
- astropy.table I/O infrastructure for custom readers/writers implemented. [#305]
- Added support for reading/writing HDF5 files [#461]
- Added support for masked tables with missing or invalid data [#451]
- New `astropy.time` sub-package. [#332]
- New `astropy.units` sub-package that includes a class for units (`astropy.units.Unit`) and scalar quantities that have units (`astropy.units.Quantity`). [#370, #445]

  This has the following effects on other sub-packages:
- In `astropy.wcs`, the `wcs.cunit` list now takes and returns `astropy.units.Unit` objects. [#379]
- In `astropy.nddata`, units are now stored as `astropy.units.Unit` objects. [#382]
- In `astropy.table`, units on columns are now stored as `astropy.units.Unit` objects. [#380]
- In `astropy.constants`, constants are now stored as `astropy.units.Quantity` objects. [#529]

astropy.io.ascii

- Improved integration with the `astropy.table` Table class so that table and column metadata (e.g. keywords, units, description, formatting) are directly available in the output table object. The CDS, DAOphot, and IPAC format readers now provide this type of integrated metadata.

- Changed to using `astropy.table` masked tables instead of NumPy masked arrays for tables with missing values.
- Added SExtractor table reader to `astropy.io.ascii` [#420]
- Removed the Memory reader class which was used to convert data input passed to the `write` function into an internal table. Instead `write` instantiates an astropy Table object using the data input to `write`.
- Removed the NumpyOutputter as the output of reading a table is now always a `Table` object.
- Removed the option of supplying a function as a column output formatter.
- Added a new `strip_whitespace` keyword argument to the `write` function. This controls whether whitespace is stripped from the left and right sides of table elements before writing. Default is True.
- Fixed a bug in reading IPAC tables with null values.
- Generalized I/O infrastructure so that `astropy.nddata` can also have custom readers/writers [#659]

astropy.wcs

- From updating the underlying wcslib 4.16:
- When `astropy.wcs.WCS` constructs a default coordinate representation it will give it the special name “DEFAULTS”, and will not report “Found one coordinate representation”.

Other Changes and Additions

- A configuration file with all options set to their defaults is now generated when astropy is installed. This file will be pulled in as the users’ astropy configuration file the first time they `import astropy`. [#498]
- Astropy doc themes moved into `astropy.sphinx` to allow affiliated packages to access them.
- Added expanded documentation for the `astropy.cosmology` sub-package. [#272]
- Added option to disable building of “legacy” packages (pyfits, vo, etc.).
- The value of the astronomical unit (au) has been updated to that adopted by IAU 2012 Resolution B2, and the values of the pc and kpc constants have been updated to reflect this. [#368]
- Added links to the documentation pages to directly edit the documentation on GitHub. [#347]
- Several updates merged from `pywcs` into `astropy.wcs` [#384]:
- Improved the reading of distortion images.
- Added a new option to choose whether or not to write SIP coefficients.
- Uses the `relax` option by default so that non-standard keywords are allowed. [#585]

- Added HTML representation of tables in IPython notebook [#409]
- Rewrote CFITSIO-based backend for handling tile compression of FITS files. It now uses a standard CFITSIO instead of heavily modified pieces of CFITSIO as before. Astropy ships with its own copy of CFITSIO v3.30, but system packagers may choose instead to strip this out in favor of a system-installed version of CFITSIO. This corresponds to PyFITS ticket 169. [#318]
- Moved `astropy.config.data` to `astropy.utils.data` and re-factored the I/O routines to separate out the generic I/O code that can be used to open any file or resource from the code used to access Astropy-related data. The 'core' I/O routine is now `get_readable_fileobj`, which can be used to access any local as well as remote data, supports caching, and can decompress gzip and bzip2 files on-the-fly. [#425]
- Added a classmethod to `astropy.coordinates.coordsystems.SphericalCoordinatesBase` that performs a name resolve query using Sesame to retrieve coordinates for the requested object. This works for any subclass of `SphericalCoordinatesBase`, but requires an internet connection. [#556]
- astropy.nddata.convolution removed requirement of PyFFTW3; uses Numpy's FFT by default instead with the added ability to specify an FFT implementation to use. [#660]

Bug Fixes

astropy.io.ascii

- Fixed crash when pprinting a row with INDEF values. [#511]
- Fixed failure when reading DAOphot files with empty keyword values. [#666]

astropy.io.fits

- Improved handling of scaled images and pseudo-unsigned integer images in compressed image HDUs. They now work more transparently like normal image HDUs with support for the `do_not_scale_image_data` and `uint` options, as well as `scale_back` and `save_backup`. The `.scale()` method works better too. Corresponds to PyFITS ticket 88.
- Permits non-string values for the EXTNAME keyword when reading in a file, rather than throwing an exception due to the malformatting. Added verification for the format of the EXTNAME keyword when writing. Corresponds to PyFITS ticket 96.
- Added support for EXTNAME and EXTVER in PRIMARY HDUs. That is, if EXTNAME is specified in the header, it will also be reflected in the `.name` attribute and in `fits.info()`. These keywords used to be verboten in PRIMARY HDUs, but the latest version of the FITS standard allows them.

Corresponds to PyFITS ticket 151.

- HCOMPRESS can again be used to compress data cubes (and higher-dimensional arrays) so long as the tile size is effectively 2-dimensional. In fact, compatible tile sizes will automatically be used even if they're not explicitly specified. Corresponds to PyFITS ticket 171.
- Fixed a bug that could cause a deadlock in the filesystem on OSX when reading the data from certain types of FITS files. This only occurred when used in conjunction with Numpy 1.7. [#369]
- Added support for the optional `endcard` parameter in the `Header.fromtextfile()` and `Header.totextfile()` methods. Although `endcard=False` was a reasonable default assumption, there are still text dumps of FITS headers that include the END card, so this should have been more flexible. Corresponds to PyFITS ticket 176.
- Fixed a crash when running fitsdiff on two empty (that is, zero row) tables. Corresponds to PyFITS ticket 178.
- Fixed an issue where opening a FITS file containing a random group HDU in update mode could result in an unnecessary rewriting of the file even if no changes were made. This corresponds to PyFITS ticket 179.
- Fixed a crash when generating diff reports from diffs using the `ignore_comments` options. Corresponds to PyFITS ticket 181.
- Fixed some bugs with WCS distortion paper record-valued keyword cards:
- Cards that looked kind of like RVKCs but were not intended to be were over-permissively treated as such–commentary keywords like COMMENT and HISTORY were particularly affected. Corresponds to PyFITS ticket 183.
- Looking up a card in a header by its standard FITS keyword only should always return the raw value of that card. That way cards containing values that happen to valid RVKCs but were not intended to be will still be treated like normal cards. Corresponds to PyFITS ticket 184.
- Looking up a RVKC in a header with only part of the field-specifier (for example "DP1.AXIS" instead of "DP1.AXIS.1") was implicitly treated as a wildcard lookup. Corresponds to PyFITS ticket 184.
- Fixed a crash when diffing two FITS files where at least one contains a compressed image HDU which was not recognized as an image instead of a table. Corresponds to PyFITS ticket 187.
- Fixed a bug where opening a file containing compressed image HDUs in 'update' mode and then immediately closing it without making any changes caused the file to be rewritten unnecessarily.
- Fixed two memory leaks that could occur when writing compressed image data, or in some cases when opening files containing compressed image HDUs in 'update' mode.
- Fixed a bug where `ImageHDU.scale(option='old')` wasn't working at all–it was not restoring the image to its original BSCALE and BZERO values.
- Fixed a bug when writing out files containing zero-width table columns,

where the TFIELDS keyword would be updated incorrectly, leaving the table largely unreadable.
- Fixed a minor string formatting issue.
- Fixed bugs in the backwards compatibility layer for the `CardList.index` and `CardList.count` methods. Corresponds to PyFITS ticket 190.
- Improved `__repr__` and text file representation of cards with long values that are split into CONTINUE cards. Corresponds to PyFITS ticket 193.
- Fixed a crash when trying to assign a long (> 72 character) value to blank (‘’) keywords. This also changed how blank keywords are represented–there are still exactly 8 spaces before any commentary content can begin; this *may* affect the exact display of header cards that assumed there could be fewer spaces in a blank keyword card before the content begins. However, the current approach is more in line with the requirements of the FITS standard. Corresponds to PyFITS ticket 194.

astropy.io.votable

- The `Table` class now maintains a single array object which is a Numpy masked array. For variable-length columns, the object that is stored there is also a Numpy masked array.
- Changed the `pedantic` configuration option to be `False` by default due to the vast proliferation of non-compliant VO Tables. [#296]
- Renamed `astropy.io.vo` to `astropy.io.votable`.

astropy.table

- Added a workaround for an upstream bug in Numpy 1.6.2 that could cause a maximum recursion depth RuntimeError when printing table rows. [#341]

astropy.wcs

- Updated to wcslib 4.15 [#418]
- Fixed a problem with handling FITS headers on locales that do not use dot as a decimal separator. This required an upstream fix to wcslib which is included in wcslib 4.14. [#313]
- Fixed some tests that could fail due to missing/incorrect logging configuration–ensures that tests don’t have any impact on the default log location or contents. [#291]
- Various minor documentation fixes [#293 and others]
- Fixed a bug where running the tests with the `py.test` command still tried to replace the system-installed pytest with the one bundled with Astropy. [#454]
- Improved multiprocessing compatibility for file downloads. [#615]
- Fixed handling of Cython modules when building from a source checkout of

a tagged release version. [#594]
- Added a workaround for a bug in Sphinx that could occur when using the `:tocdepth:` directive. [#595]
- Minor VOTable fixes [#596]
- Fixed how `setup.py` uses `distribute_setup.py` to prevent possible `VersionConflict` errors when an older version of distribute is already installed on the user’s system. [#616, #640]
- Changed use of `log.warn` in the logging module to `log.warning` since the former is deprecated. [#624]

0.1 (2012-06-19)

- Initial release.

There are some additional tools, mostly of use for maintainers, in the astropy/astropy-procedures repository.

Project details

Current status of sub-packages

Astropy has benefited from the addition of widely tested legacy code, as well as new development, resulting in variations in stability across sub-packages. This document summarizes the current status of the Astropy sub-packages, so that users understand where they might expect changes in future, and which sub-packages they can safely use for production code.

The classification is as follows:

⚫ Planned

🟠 Actively developed, be prepared for possible significant changes.

🔵 Reasonably stable, any significant changes/additions will generally include backwards-compatiblity.

🟢 Mature. Additions/improvements possible, but no major changes planned.

🟤 Pending deprecation. Might be deprecated in a future version.

🔴 Deprecated. Might be removed in a future version.

The current planned and existing sub-packages are:

Sub-Package		Comments
astropy.config	🟢	Configuration received major overhaul in v0.4. Since then on, the package has been stable.
astropy.constants	🟢	The package has been stable except for the occasional additions of new constants. Since v3.0, it includes the ability to use sets of constants from previous versions.

astropy.convolution	🟢	New top-level package in v0.3 (was previously part of `astropy.nddata`). A major consistency improvement between fft/non-fft convolution, which is not fully backward-compatible, was added in 2.0.
astropy.coordinates	🔵	New in v0.2, major changes in v0.4. Subsequent versions should maintain a stable/backwards-compatible API, following the plan of APE 5. Further major additions/enhancements likely, but with basic framework unchanged.
astropy.cosmology	🟢	Incremental improvements since v0.1, stable API last several versions.
astropy.io.ascii	🟢	Originally developed as `asciitable`, and has maintained a stable API.
astropy.io.fits	🟢	Originally developed as `pyfits`, and retains an API consistent with the standalone version.
astropy.io.misc	🟢	The functionality that is currently present is stable, but this sub-package will likely see major additions in future.
astropy.io.votable	🟢	Originally developed as `vo.table`, and has a stable API.
astropy.modeling	🟠	New in v0.3. Major changes in v1.0, significant additions planned. Backwards-compatibility likely to be maintained, but not guaranteed.
astropy.nddata	🟠	Significantly revised in v1.0 to implement APE 7. Major changes in the API are not anticipated, broader use may reveal flaws that require API changes.
astropy.samp	🟢	Virtual Observatory service access: SAMP. This was renamed from astropy.vo.samp to astropy.samp in 2.0.
astropy.stats	🟠	Likely to maintain backwards-compatibility, but functionality continually being expanded, so significant additions likely in the future.
astropy.table	🟢	Incremental improvements since v0.1, mostly stable API with backwards compatibility an explicit goal.
astropy.time	🟢	Incremental improvements since v0.1, API likely to remain stable for the foreseeable future.
astropy.timeseries	🟠	New in v3.2, in heavy development.
astropy.units	🟢	Incremental improvements since v0.4. Functionality mature and unlikely to change. Efforts focused on performance and increased interoperability with Numpy functions.
astropy.utils	🟠	Contains mostly utilities destined for internal use with other parts of Astropy. Existing functionality generally stable, but regular additions and occasional changes.
astropy.uncertainty	🟠	New in v3.1, in development.
astropy.visualization	🟠	New in v1.0, and in development.
astropy.visualization.wcsaxes	🔵	New in v1.3. Originally developed as `wcsaxes` and has maintained a stable API.
astropy.wcs	🔵	Originally developed as `pywcs`, and has a stable API for now. However, there are plans to generalize the WCS interface to accommodate non-FITS WCS transformations, and this may lead to small changes in the user interface.

Major Release History

Examples in these documents are frozen in time to respect the status of the API at the time of the release they are describing. Please refer to the main, up-to-date documentation if you run into any issues with the functionality highlighted in these pages.

- What's New in Astropy 4.1?

- What's New in Astropy 4.0?
- What's New in Astropy 3.2?
- What's New in Astropy 3.1?
- What's New in Astropy 3.0?
- What's New in Astropy 2.0?
- What's New in Astropy 1.3?
- What's New in Astropy 1.2?
- What's New in Astropy 1.1?
- What's New in Astropy 1.0?
- What's New in Astropy 0.4?
- What's New in Astropy 0.3?
- What's New in Astropy 0.2?
- What's New in Astropy 0.1?

Known Issues

While most bugs and issues are managed using the astropy issue tracker, this document lists issues that are too difficult to fix, may require some intervention from the user to work around, or are caused by bugs in other projects or packages.

Issues listed on this page are grouped into two categories: The first is known issues and shortcomings in actual algorithms and interfaces that currently do not have fixes or workarounds, and that users should be aware of when writing code that uses `astropy`. Some of those issues are still platform-specific, while others are very general. The second category is of common issues that come up when configuring, building, or installing `astropy`. This also includes cases where the test suite can report false negatives depending on the context/ platform on which it was run.

Known Deficiencies

Quantities Lose Their Units with Some Operations

Quantities are subclassed from NumPy's **`ndarray`** and in some NumPy operations (and in SciPy operations using NumPy internally) the subclass is ignored, which means that either a plain array is returned, or a **`Quantity`** without units. E.g., prior to astropy 4.0 and numpy 1.17:

```
>>> import astropy.units as u
>>> import numpy as np
>>> q = u.Quantity(np.arange(10.), u.m)
>>> np.dot(q,q)
285.0
>>> np.hstack((q,q))
<Quantity [0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 0., 1., 2., 3.,
4., 5.,
           6., 7., 8., 9.] (Unit not initialised)>
```

And for all versions:

```
>>> ratio = (3600 * u.s) / (1 * u.h)
>>> ratio
<Quantity 3600. s / h>
>>> np.array(ratio)
array(3600.)
>>> np.array([ratio])
array([1.])
```

Workarounds are available for some cases. For the above:

```
>>> q.dot(q)
<Quantity 285. m2>

>>> np.array(ratio.to(u.dimensionless_unscaled))
array(1.)

>>> u.Quantity([q, q]).flatten()
<Quantity [0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 0., 1., 2., 3.,
4., 5.,
            6., 7., 8., 9.] m>
```

An incomplete list of specific functions which are known to exhibit this behavior (prior to astropy 4.0 and numpy 1.17) follows:

- **numpy.dot**
- **numpy.hstack**, **numpy.vstack**, `numpy.c_`, `numpy.r_`, **numpy.append**
- **numpy.where**
- **numpy.choose**
- **numpy.vectorize**
- pandas DataFrame(s)

See: https://github.com/astropy/astropy/issues/1274

Care must be taken when setting array slices using Quantities:

```
>>> a = np.ones(4)
>>> a[2:3] = 2*u.kg
>>> a
array([1., 1., 2., 1.])
```

```
>>> a = np.ones(4)
>>> a[2:3] = 1*u.cm/u.m
>>> a
array([1., 1., 1., 1.])
```

Either set single array entries or use lists of Quantities:

```
>>> a = np.ones(4)
>>> a[2] = 1*u.cm/u.m
>>> a
array([1.  , 1.  , 0.01, 1.  ])
```

```
>>> a = np.ones(4)
>>> a[2:3] = [1*u.cm/u.m]
>>> a
array([1.  , 1.  , 0.01, 1.  ])
```

Both will throw an exception if units do not cancel, e.g.:

```
>>> a = np.ones(4)
>>> a[2] = 1*u.cm
Traceback (most recent call last):
...
```

```
TypeError: only dimensionless scalar quantities can be converted to
Python scalars
```

See: https://github.com/astropy/astropy/issues/7582

Numpy array creation functions cannot be used to initialize Quantity

Trying the following example will throw an UnitConversionError on NumPy before version 1.20 and ignore the unit in later versions:

```
>>> my_quantity = u.Quantity(1, u.m)
>>> np.full(10, my_quantity)
Traceback (most recent call last):
...
UnitConversionError: 'm' (length) and '' (dimensionless) are not
convertible
```

A workaround for this at the moment would be to do:

```
>>> np.full(10, 1) << u.m
<Quantity [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.] m>
```

As well as with **`full`** one cannot do **`zeros`**, **`ones`**, and **`empty`**.

Quantities Lose Their Units When Broadcasted

When broadcasting Quantities, it is necessary to pass `subok=True` to **`broadcast_to`**, or else a bare **`ndarray`** will be returned:

```
>>> q = u.Quantity(np.arange(10.), u.m)
>>> b = np.broadcast_to(q, (2, len(q)))
>>> b
array([[0., 1., 2., 3., 4., 5., 6., 7., 8., 9.],
       [0., 1., 2., 3., 4., 5., 6., 7., 8., 9.]])
>>> b2 = np.broadcast_to(q, (2, len(q)), subok=True)
>>> b2
<Quantity [[0., 1., 2., 3., 4., 5., 6., 7., 8., 9.],
           [0., 1., 2., 3., 4., 5., 6., 7., 8., 9.]] m>
```

This is analogous to the case of passing a Quantity to **`array`**:

```
>>> a = np.array(q)
>>> a
array([0., 1., 2., 3., 4., 5., 6., 7., 8., 9.])
>>> a2 = np.array(q, subok=True)
>>> a2
<Quantity [0., 1., 2., 3., 4., 5., 6., 7., 8., 9.] m>
```

See: https://github.com/astropy/astropy/issues/7832

Quantities Float Comparison with np.isclose Fails

Comparing Quantities floats using the NumPy function **isclose** fails on NumPy versions before 1.17 as the comparison between `a` and `b` is made using the formula

\[|a - b| \le (a_\textrm{tol} + r_\textrm{tol} \times |b|)\]

This will result in the following traceback when using this with Quantities:

```
>>> from astropy import units as u, constants as const
>>> import numpy as np
>>> np.isclose(500 * u.km/u.s, 300 * u.km / u.s)
Traceback (most recent call last):
...
UnitConversionError: Can only apply 'add' function to dimensionless
quantities when other argument is not a quantity (unless the latter
is all zero/infinity/nan)
```

If one cannot upgrade to numpy 1.17 or later, one solution is:

```
>>> np.isclose(500 * u.km/u.s, 300 * u.km / u.s, atol=1e-8 * u.mm /
u.s)
False
```

Quantities in np.linspace Failure on NumPy 1.10

linspace does not work correctly with quantities when using NumPy 1.10.0 to 1.10.5 due to a bug in NumPy. The solution is to upgrade to NumPy 1.10.6 or later, in which the bug was fixed.

mmap Support for `astropy.io.fits` on GNU Hurd

On Hurd and possibly other platforms, `flush()` on memory-mapped files are not implemented, so writing changes to a mmap'd FITS file may not be reliable and is thus disabled. Attempting to open a FITS file in writeable mode with mmap will result in a warning (and mmap will be disabled on the file automatically).

See: https://github.com/astropy/astropy/issues/968

Bug with Unicode Endianness in `io.fits` for Big Endian

Processors

On big endian processors (e.g. SPARC, PowerPC, MIPS), string columns in FITS files may not be correctly read when using the `Table.read` interface. This will be fixed in a subsequent bug fix release of `astropy` (see bug report here).

Color Printing on Windows

Colored printing of log messages and other colored text does work in Windows, but only when running in the IPython console. Colors are not currently supported in the basic Python command-line interpreter on Windows.

`numpy.int64` does not decompose input `Quantity` objects

Python's `int()` goes through `__index__` while `numpy.int64` or `numpy.int_` do not go through `__index__`. This means that an upstream fix in `numpy` is required in order for ``astropy.units` to control decomposing the input in these functions:

```
>>> np.int64((15 * u.km) / (15 * u.imperial.foot))
1
>>> np.int_((15 * u.km) / (15 * u.imperial.foot))
1
>>> int((15 * u.km) / (15 * u.imperial.foot))
3280
```

To convert a dimensionless **`Quantity`** to an integer, it is therefore recommended to use `int(...)`.

Inconsistent behavior when converting complex numbers to floats

Attempting to use **`float`** or NumPy's `numpy.float` on a standard complex number (e.g., `5 + 6j`) results in a **`TypeError`**. In contrast, using **`float`** or `numpy.float` on a complex number from NumPy (e.g., `numpy.complex128`) drops the imaginary component and issues a `numpy.ComplexWarning`. This inconsistency persists between **`Quantity`** instances based on standard and NumPy complex numbers. To get the real part of a complex number, it is recommended to use `numpy.real`.

Build/Installation/Test Issues

Anaconda Users Should Upgrade with `conda`, Not `pip`

Upgrading `astropy` in the Anaconda Python distribution using `pip` can result in a corrupted install with a mix of files from the old version and the new version. Anaconda users should update with `conda update astropy`. There may be a brief delay between the release of `astropy` on PyPI and its release via the `conda` package manager; users can check the availability of new versions with `conda search astropy`.

Locale Errors in MacOS X and Linux

On MacOS X, you may see the following error when running `pip`:

```
...
ValueError: unknown locale: UTF-8
```

This is due to the `LC_CTYPE` environment variable being incorrectly set to `UTF-8` by default, which is not a valid locale setting.

On MacOS X or Linux (or other platforms) you may also encounter the following error:

```
...
  stderr = stderr.decode(stdio_encoding)
TypeError: decode() argument 1 must be str, not None
```

This also indicates that your locale is not set correctly.

To fix either of these issues, set this environment variable, as well as the `LANG` and `LC_ALL` environment variables to e.g. `en_US.UTF-8` using, in the case of `bash`:

```
export LANG="en_US.UTF-8"
export LC_ALL="en_US.UTF-8"
export LC_CTYPE="en_US.UTF-8"
```

To avoid any issues in future, you should add this line to your e.g. `~/.bash_profile` or `.bashrc` file.

To test these changes, open a new terminal and type `locale`, and you should see something like:

```
$ locale
LANG="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_CTYPE="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
```

```
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_ALL="en_US.UTF-8"
```

If so, you can go ahead and try running `pip` again (in the new terminal).

Failing Logging Tests When Running the Tests in IPython

When running the Astropy tests using `astropy.test()` in an IPython interpreter, some of the tests in the `astropy/tests/test_logger.py` *might* fail depending on the version of IPython or other factors. This is due to mutually incompatible behaviors in IPython and pytest, and is not due to a problem with the test itself or the feature being tested.

See: https://github.com/astropy/astropy/issues/717

Some Docstrings Can Not Be Displayed in IPython < 0.13.2

Displaying long docstrings that contain Unicode characters may fail on some platforms in the IPython console (prior to IPython version 0.13.2):

```
In [1]: import astropy.units as u

In [2]: u.Angstrom?
Out[2]: ERROR: UnicodeEncodeError: 'ascii' codec can't encode
character u'\xe5' in
position 184: ordinal not in range(128) [IPython.core.page]
```

This can be worked around by changing the default encoding to `utf-8` by adding the following to your `sitecustomize.py` file:

```
import sys
sys.setdefaultencoding('utf-8')
```

Note that in general, this is not recommended, because it can hide other Unicode encoding bugs in your application. However, if your application does not deal with text processing and you just want docstrings to work, this may be acceptable.

The IPython issue: https://github.com/ipython/ipython/pull/2738

Compatibility Issues with pytest 3.7 and later

Due to a bug in pytest related to test collection, the tests for the core `astropy` package for version 2.0.x (LTS), and for packages using the core package's test infrastructure and being tested against 2.0.x (LTS), will not be executed correctly with pytest 3.7, 3.8, or 3.9. The symptom of this bug is that no tests or only tests in RST files are collected. In addition, `astropy` 2.0.x (LTS) is not compatible with pytest 4.0 and above, as in this case deprecation errors from pytest can cause tests to fail. Therefore, when testing against `astropy` v2.0.x (LTS), pytest 3.6 or earlier versions should be used. These issues do not occur in version 3.0.x and above of the core package.

There is an unrelated issue that also affects more recent versions of `astropy` when testing with pytest 4.0 and later, which can cause issues when collecting tests — in this case, the symptom is that the test collection hangs and/or appears to run the tests recursively. If you are maintaining a package that was created using the Astropy package template, then this can be fixed by updating to the latest version of the `_astropy_init.py` file. The root cause of this issue is that pytest now tries to pick up the top-level `test()` function as a test, so we need to make sure that we set a `test.__test__` attribute on the function to `False`.

Authors and Credits

Astropy Project Coordinators

- Tom Aldcroft
- Kelle Cruz
- Thomas Robitaille
- Erik Tollerud

Core Package Contributors

- Aaron Meisner
- Aarya Patil
- Abhinuv Nitin Pitale
- Abigail Stevens
- Adam Ginsburg
- Adele Plunkett
- Aditya Sharma
- Adrian Price-Whelan
- Akash Deshpande
- Al Niessner
- Albert Y. Shih
- Aleh Khvalko
- Alex Conley
- Alex Drlica-Wagner
- Alex Hagen
- Alex Rudy

- Alex de la Vega
- Alexander Bakanov
- Alexandre Beelen
- Amit Kumar
- Ana Posses
- Anany Shrey Jain
- Anchit Jain
- Andreas Baumbach
- Andrew Hearin
- Aniket Kulkarni
- Anirudh Katipally
- Anne Archibald
- Antetokounpo
- Anthony Horton
- Antony Lee
- Arfon Smith
- Arie Kurniawan
- Arne de Laat
- Arthur Eigenbrot
- Asish Panda
- Asra Nizami
- Austen Groener
- Axel Donath
- Azalee Bostroem
- Benjamin Alan Weaver
- Benjamin Roulston
- Benjamin Winkel
- Bernardo Sulzbach
- Bernie Simon
- Bili Dong
- Bogdan Nicula
- Brett Morris
- Brigitta Sipőcz
- Bruno Oliveira
- Bryce Kalmbach
- Bryce Nordgren
- Carl Osterwisch
- Carl Schaffer
- Chris Beaumont
- Chris Hanley
- Chris Osborne
- Chris Simpson
- Christian Clauss
- Christian Hettlage

- Christoph Deil
- Christoph Gohlke
- Christopher Bonnett
- Clara Brasseur
- Clare Shanahan
- Clément Robert
- Cristian Ardelean
- Curtis McCully
- Dan Foreman-Mackey
- Dan P. Cunningham
- Dan Taranu
- Daniel Bell
- Daniel D’Avella
- Daniel Datsev
- Daniel Lenz
- Daniel Ruschel Dutra
- Danny Goldstein
- Daria Cara
- David Kirkby
- David M. Palmer
- David Pérez-Suárez
- David Shiga
- David Shupe
- David Stansby
- Demitri Muna
- Derek Homeier
- Devin Crichton
- Dominik Klaes
- Doug Burke
- Drew Leonard
- Duncan Macleod
- Dylan Gregersen
- Ed Slavich
- Edward Betts
- Eli Bressert
- Elijah Bernstein-Cooper
- Eloy Salinas
- Emily Deibert
- Emma Hogan
- Eric Depagne
- Eric Jeschke
- Eric Koch
- Erik M. Bray
- Erik Tollerud

- Erin Allard
- Esteban Pardo Sánchez
- Evert Rol
- Felix Yan
- Francesco Biscani
- Francesco Montanari
- Francesco Montesano
- Frédéric Chapoton
- Frédéric Grollier
- Gabriel Brammer
- Gabriel Perren
- Geert Barentsen
- Georgiana Ogrean
- Gerrit Schellenberger
- Giang Nguyen
- Giorgio Calderone
- Graham Kanarek
- Grant Jenks
- Gregory Dubois-Felsmann
- Griffin Hosseinzadeh
- Gustavo Bragança
- Hannes Breytenbach
- Hans Moritz Günther
- Harry Ferguson
- Helen Sherwood-Taylor
- Himanshu Pathak
- Hugo Buddelmeijer
- Humna Awan
- J. Goutin
- J. Xavier Prochaska
- JC Hsu
- Jake VanderPlas
- James Davies
- James Dearman
- James Noss
- James Taylor
- James Turner
- Jan Skowron
- Jane Rigby
- Jani Šumak
- Javier Pascual Granado
- Jean Connelly
- Jeff Taylor
- Jeffrey McBeth

- Jerry Ma
- Joanna Power
- Joe Hunkeler
- Joe Lyman
- Joe Philip Ninan
- John Fisher
- John Parejko
- Johnny Greco
- Jonas Große Sundrup
- Jonathan Eisenhamer
- Jonathan Foster
- Jonathan Sick
- Jonathan Whitmore
- Jörg Dietrich
- Joseph Jon Booker
- Joseph Long
- Joseph Ryan
- Joseph Schlitz
- José Sabater Montes
- Juan Luis Cano Rodríguez
- Juanjo Bazán
- Julien Woillez
- Jurien Huisman
- Kacper Kowalik
- Karan Grover
- Karl Gordon
- Karl Vyhmeister
- Katrin Leinweber
- Kelle Cruz
- Kevin Gullikson
- Kevin Sooley
- Kewei Li
- Kieran Leschinski
- Kirill Tchernyshyov
- Kris Stern
- Kristin Berry
- Kyle Barbary
- Kyle Oman
- Larry Bradley
- Laura Watkins
- Lauren Glattly
- Leah Fulmer
- Lehman Garrison
- Lennard Kiehl

- Leo Singer
- Leonardo Ferreira
- Lia Corrales
- Lingyi Hu
- Lisa Martin
- Lisa Walter
- Luigi Paioro
- Luke G. Bouma
- Luke Kelley
- M Atakan Gürkan
- M S R Dinesh
- Mabry Cervin
- Madhura Parikh
- Magali Mebsout
- Manas Satish Bedmutha
- Maneesh Yadav
- Mangala Gowri Krishnamoorthy
- Manish Biswas
- Manodeep Sinha
- Mark Fardal
- Mark Taylor
- Marten van Kerkwijk
- Martin Glatzle
- Matej Stuchlik
- Mathieu Servillat
- Matt Davis
- Matteo Bachetti
- Matthew Bourque
- Matthew Brett
- Matthew Craig
- Matthew Petroff
- Matthew Turk
- Mavani Bhautik
- Max Silbiger
- Max Voronkov
- Maximilian Nöthe
- Médéric Boquien
- Megan Sosey
- Michael Droettboom
- Michael Hirsch
- Michael Hoenig
- Michael Lindner-D'Addario
- Michael Mueller
- Michael Seifert

- Michael Wood-Vasey
- Michael Zhang
- Michele Costa
- Michele Mastropietro
- Miguel de Val-Borro
- Mihai Cara
- Mike Alexandersen
- Mike McCarty
- Mikhail Minin
- Mikołaj
- Miruna Oprescu
- Moataz Hisham
- Mohan Agrawal
- Molly Peeples
- Nabil Freij
- Nadia Dencheva
- Nathanial Hendler
- Nathaniel Starkman
- Neal McBurnett
- Neil Crighton
- Neil Parley
- Nicholas Earl
- Nicholas S. Kern
- Nicholas Saunders
- Nick Lloyd
- Nick Murphy
- Nimit Bhardwaj
- Noah Zuckman
- Nora Luetzgendorf
- Ole Streicher
- Orion Poplawski
- Parikshit Sakurikar
- Patricio Rojo
- Patti Carroll
- Paul Barrett
- Paul Hirst
- Paul Price
- Paul Sladen
- Pauline Barmby
- Perry Greenfield
- Peter Cock
- Peter Teuben
- Pey Lian Lim
- Prasanth Nair

- Pratik Patel
- Pritish Chakraborty
- Ralf Gommers
- Rashid Khan
- Rasmus Handberg
- Ray Plante
- Régis Terrier
- Ricardo Ogando
- Ricky O’Steen
- Ritiek Malhotra
- Ritwick DSouza
- Roban Hultman Kramer
- Robel Geda
- Robert Cross
- Rocio Kiman
- Rohan Rajpal
- Rohit Kapoor
- Rohit Patil
- Roman Tolesnikov
- Rui Xue
- Ryan Abernathey
- Ryan Cooke
- Ryan Fox
- Sadie Bartholomew
- Sam Verstocken
- Samuel Brice
- Sanjeev Dubey
- Sara Ogaz
- Sarah Kendrew
- Sashank Mishra
- Saurav Sachidanand
- Scott Thomas
- Semyeong Oh
- Serge Montagnac
- Sergio Pascual
- SF Graves
- Shailesh Ahuja
- Shantanu Srivastava
- Shilpi Jain
- Shivan Sornarajah
- Shivansh Mishra
- Shresth Verma
- Shreyas Bapat
- Sigurd Næss

- Simon Conseil
- Simon Gibbons
- Simon Liedtke
- Simon Torres
- Sourabh Cheedella
- Srikrishna Sekhar
- Stefan Becker
- Stefan Nelson
- Stephen Portillo
- Steve Crawford
- Steven Bamford
- Stuart Littlefair
- Stuart Mumford
- Sudheesh Singanamalla
- Sushobhana Patra
- Swapnil Sharma
- T. Carl Beery
- Tanuj Rastogi
- Thomas Erben
- Thomas Robitaille
- Thompson Le Blanc
- Tiffany Jansen
- Tim Jenness
- Tim Plummer
- Tito Dal Canton
- Tom Aldcroft
- Tom Donaldson
- Tom J Wilson
- Tom Kooij
- Tomas Babej
- Tyler Finethy
- VSN Reddy Janga
- Vatsala Swaroop
- Vinayak Mehta
- Vishnunarayan K I
- Vital Fernández
- Víctor Terrón
- Víctor Zabalza
- Wilfred Tyler Gee
- Wolfgang Kerzendorf
- Yannick Copin
- Yash Kumar
- Yash Sharma
- Yingqi Ying

- Zach Edwards
- Zachary Kurtz
- Zeljko Ivezic
- Zlatan Vasović
- Zé Vinicius

Other Credits

- Kyle Barbary for designing the Astropy logos and documentation themes.
- Andrew Pontzen and the pynbody team (For code that grew into **`astropy.units`**)
- Everyone on the astropy-dev mailing list and the Astropy mailing list for contributing to many discussions and decisions!

(If you have contributed to the `astropy` core package and your name is missing, please send an email to the coordinators, or open a pull request for this page in the astropy repository)

For how to acknowledge Astropy, please see the Acknowledging or Citing Astropy page.

Licenses

Astropy License

Other Licenses

Full licenses for third-party software astropy is derived from or included with Astropy can be found in the `'licenses/'` directory of the source code distribution.

Index

- Index
- Module Index
- Search Page