Astropy Documentation

Release 0.2.dev2728

The Astropy Developers

June 30, 2013

CONTENTS

Astropy is a community-driven package intended to contain much of the core functionality and some common tools needed for performing astronomy and astrophysics with Python.

Part I

User Documentation

CHAPTER

ONE

WHAT'S NEW IN ASTROPY 0.2

1.1 Overview

Here we describe a broad overview of the Astropy project and its parts.

1.1.1 Astropy Project Concept

The "Astropy Project" is distinct from the `astropy` package. The Astropy Project is a process intended to facilitate communication and interoperability of python packages/codes in astronomy and astrophysics. The project thus encompasses the `astropy` core package (which provides a common framework), all "affiliated packages" (described below in `Affiliated Packages`), and a general community aimed at bringing resources together and not duplicating efforts.

1.1.2 `astropy` Core Package

The `astropy` package (alternatively known as the "core" package) contains various classes, utilities, and a packaging framework intended to provide commonly-used astronomy tools. It is divided into a variety of sub-packages, which are documented in the remainder of this documentation (see *User Documentation* for documentation of these components).

The core also provides this documentation, and a variety of utilities that simplify starting other python astronomy/astrophysics packages. As described in the following section, these simplify the process of creating affiliated packages.

1.1.3 Affiliated Packages

The Astropy project includes the concept of "affiliated packages." An affiliated package is an astronomy-related python package that is not part of the `astropy` core source code, but has requested to be included in the Astropy project. Such a package may be a candidate for eventual inclusion in the main `astropy` package (although this is not required).

There is a registry of affiliated packages at http://affiliated.astropy.org, and in the near future, the `astropy` core will include a tool to install affiliated packages by name. See the `config` module documentation for details about the affiliated package registry and install tool. Note that affiliated packages do not use the `astropy` namespace, which is reserved for the core. Instead, they either use their package name directly, or `awastropy.packagename` ("affiliated with astropy").

If you are interested in starting an affiliated package, or have a package you are interested in making more compatible with astropy, the `astropy` core package includes a variety of features that simplify and homogenize package management. Astropy provides a package template that provides a common way to organize packages, to make your life

simpler. You can use this template either with a new package you are starting or an existing package to make it more compatible with Astropy and the affiliated package installer. See the usage instructions in the template for further details.

1.1.4 Community

Aside from the actual code, Astropy is also a community of astronomy-associated users and developers that agree that sharing utilities is healthy for the community and the science it produces. This community is of course central to accomplishing anything with the code itself.

1.2 Installation

1.2.1 Requirements

Astropy has the following strict requirements:

- Python 2.6, 2.7, 3.1 or 3.2
- Numpy 1.4 or later

Astropy also depends on other packages for optional features:

- h5py: To read/write `Table` objects from/to HDF5 files
- scipy: To power a variety of features (currently mainly cosmology-related functionality)
- xmllint: To validate VOTABLE XML files.

However, note that these only need to be installed if those particular features are needed. Astropy will import even if these dependencies are not installed.

1.2.2 Installing Astropy

Using `pip`

To install Astropy with `pip`, simply run:

```
pip install astropy
```

Binary installers

No binary installers are available at this time.

Testing Astropy

The easiest way to test 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.

1.2.3 Building from source

Prerequisites

You will need a compiler suite and the development headers for Python and Numpy in order to build Astropy. Using the package manager for your platform will usually be the easiest route.

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

You will also need Cython installed to build from source, unless you are installing a numbered release. (The releases packages have the necessary C files packaged with them, and hence do not require Cython.)

Note: If you are using MacOS X, the easiest way to install a compiler suite is to install the MacOS X developer tools (XCode) As of XCode 4.3, the command-line compilers are no longer installed by default: you will need to open the XCode application, go to **Preferences**, then **Downloads**, and then under **Components**, click on the Install button to the right of **Command Line Tools**.

Obtaining the source packages

Source packages

Source tarballs of past releases and the current development branch of astropy can be downloaded from https://github.com/astropy/astropy/downloads

Development repository

The latest development version of Astropy can be cloned from github using this command:

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

Note: If you wish to participate in the development of Astropy, see *Developer Documentation*. This document covers only the basics necessary to install Astropy.

Building and Installing

Astropy uses the Python distutils framework for building and installing.

To build Astropy (from the root of the source tree):

```
python setup.py build
```

To install Astropy (from the root of the source tree):

```
python setup.py install
```

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 pass one or more of the `--use-system-X` flags to the `setup.py build` command.

For example, to build Astropy using the system `libexpat`, use:

```
python setup.py build --use-system-expat
```

To build using all of the system libraries, use:

```
python setup.py build --use-system-libraries
```

To see which system libraries Astropy knows how to build against, use:

```
python setup.py build --help
```

As with all distutils commandline options, they may also be provided in a `setup.cfg` in the same directory as `setup.py`. For example, to use the system `libexpat`, add the following to the `setup.cfg` file:

```
[build]
use_system_expat=1
```

Compatibility packages

Warning: This feature is still experimental, and you may run into unexpected issues with other packages, so we strongly recommend simply updating your code to use Astropy if possible, rather than rely on these compatibility packages.

Optionally, it is possible to install 'compatibility' packages that emulate the behavior of previous packages that have now been incorporated into Astropy. These are:

- PyFITS
- vo
- PyWCS

If you build Astropy with:

```
python setup.py build --enable-legacy
python setup.py install
```

or simply:

```
python setup.py install --enable-legacy
```

then you will be able to import these modules from your scripts as if the original packages had been installed. Using:

```
import pyfits
import vo
import pywcs
```

will then be equivalent to:

```
from astropy.io import fits as pyfits
from astropy.io import vo
from astropy import wcs as pywcs
```

In order to install the compatibility packages none of the original packages should be present.

Note: If you are interested in testing out existing code with Astropy without modifying the import statements, but don't want to uninstall existing packages, you can use virtualenv to set up a clean environment.

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 .

Building the documentation requires the Astropy source code and some additional packages:

- Sphinx (and its dependencies) 1.0 or later
- Graphviz

There are two ways to build the Astropy documentation. The most straightforward way is to execute the command (from the astropy source directory):

```
python setup.py build_sphinx
```

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`.

The above method builds the API documentation from the source code. Alternatively, you can do:

```
cd docs
make html
```

And the documentation will be generated in the same location, but using the *installed* version of Astropy.

Testing your Astropy build

The easiest way to test that your Astropy built correctly (without installing astropy) is to run this from the root of the source tree:

```
python setup.py test
```

There are also alternative methods of *Running Tests*.

1.3 Getting Started with Astropy

1.3.1 Importing Astropy

In order to encourage consistency amongst 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 is best to import the desired sub-pacakge 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.ICRSCoordinates(ra=10.68458, dec=41.26917, unit=(u.degree, u.degree))
<ICRSCoordinates RA=10.68458 deg, Dec=41.26917 deg>
```

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 to **never** import 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
```

1.4 N-dimensional datasets (`astropy.nddata`)

1.4.1 Introduction

`astropy.nddata` 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.

This subpackage also provides new convolution routines that differ from Scipy in that they offer a proper treatment of NaN values.

Note: The `NDData` class is still under development, and support for WCS and units is not yet implemented.

1.4.2 Getting started

An `NDData` object can be instantiated by passing it an n-dimensional Numpy array:

```
>>> from astropy.nddata import NDData
>>> array = np.random.random((12, 12, 12))  # a random 3-dimensional array
>>> ndd = NDData(array)
```

This object has a few attributes in common with Numpy:

```
>>> ndd.ndim
3
>>> ndd.shape
(12, 12, 12)
>>> ndd.dtype
dtype('float64')
```

The underlying Numpy array can be accessed via the `data` attribute:

```
>>> ndd.data
array([[[ 0.05621944,  0.85569765,  0.71609697, ...,  0.76049288,
...
```

Values can be masked using the `mask` attribute, which should be a boolean Numpy array with the same dimensions as the data, e.g.:

```
>>> ndd.mask = ndd.data > 0.9
```

A mask value of `True` indicates a value that should be ignored, while a mask value of `False` indicates a valid value.

Similarly, attributes are available to store generic meta-data, flags, and uncertainties, and the `NDData` class includes methods to combine datasets with arithmetic operations (which include uncertainties propagation). These are described in *NDData overview*.

1.4.3 Using `nddata`

NDData overview

Initializing

An `NDData` object can be instantiated by passing it an n-dimensional Numpy array:

```
>>> from astropy.nddata import NDData
>>> array = np.random.random((12, 12, 12))  # a random 3-dimensional array
>>> ndd = NDData(array)
```

or by passing it an `NDData` object:

```
>>> ndd1 = NDData(array)
>>> ndd2 = NDData(ndd1)
```

This object has a few attributes in common with Numpy:

```
>>> ndd.ndim
3
>>> ndd.shape
(12, 12, 12)
>>> ndd.dtype
dtype('float64')
```

The underlying Numpy array can be accessed via the `data` attribute:

```
>>> ndd.data
array([[[ 0.05621944,  0.85569765,  0.71609697, ...,  0.76049288,
...
```

Mask

Values can be masked using the `mask` attribute, which should be a boolean Numpy array with the same dimensions as the data, e.g.:

```
>>> ndd.mask = ndd.data > 0.9
```

A mask value of `True` indicates a value that should be ignored, while a mask value of `False` indicates a valid value.

Flags

Values can be assigned one or more flags. The `flags` attribute is used to store either a single Numpy array (of any type) with dimensions matching that of the data, or a `FlagCollection`, which is essentially a dictionary of Numpy arrays (of any type) with the same shape as the data. The following example demonstrates setting a single set of integer flags:

```
>>> ndd.flags = np.zeros(ndd.shape)
>>> ndd.flags[ndd.data < 0.1] = 1
>>> ndd.flags[ndd.data < 0.01] = 2
```

but one can also have multiple flag layers with different types:

```
>>> from astropy.nddata import FlagCollection
>>> ndd.flags = FlagCollection(shape=(12, 12, 12))
>>> ndd.flags['photometry'] = np.zeros(ndd.shape, dtype=str)
>>> ndd.flags['photometry'][ndd.data > 0.9] = 's'
>>> ndd.flags['cosmic_rays'] = np.zeros(ndd.shape, dtype=int)
>>> ndd.flags['cosmic_rays'][ndd.data > 0.99] = 99
```

and flags can easily be used to set the mask:

```
>>> ndd.mask = ndd.flags['cosmic_rays'] == 99
```

Uncertainties

`NDData` objects have an `uncertainty` attribute that can be used to set the uncertainty on the data values. This is done by using classes to represent the uncertainties of a given type. For example, to set standard deviation uncertainties on the pixel values, you can do:

```
>>> from astropy.nddata import StdDevUncertainty
>>> ndd.uncertainty = StdDevUncertainty(np.ones((12, 12, 12)) * 0.1)
```

Note: For information on creating your own uncertainty classes, see *Subclassing NDData and NDUncertainty*.

Arithmetic

Provided that the world coordinate system (WCS), units, and shape match, two `NDData` instances can be added or subtracted from each other, with uncertainty propagation, creating a new `NDData` object:

```
ndd3 = ndd1.add(ndd2)
ndd4 = ndd1.subtract(ndd2)
```

The purpose of the `add()` and `subtract()` methods is to allow the combination of two data objects that have common WCS, units, and shape, with consistent behavior for masks and flags, and with a framework to propagate uncertainties. These methods are intended for use by sub-classes and functions that deal with more complex combinations.

Warning: Uncertainty propagation is still experimental, and does not take into account correlated uncertainties.

Meta-data

The `NDData` class includes a `meta` attribute that defaults to an empty dictionary, and can be used to set overall meta-data for the dataset:

```
ndd.meta['exposure_time'] = 340.
ndd.meta['filter'] = 'J'
```

Elements of the meta-data dictionary can be set to any valid Python object:

```
ndd.meta['history'] = ['calibrated', 'aligned', 'flat-fielded']
```

Converting to Numpy arrays

`NDData` objects can also be easily converted to numpy arrays:

```
>>> import numpy as np
>>> arr = np.array(ndd)
>>> np.all(arr == mydataarray)
True
```

If a `mask` is defined, this will result in a `MaskedArray`, so in all cases a useable `numpy.ndarray` or subclass will result. This allows straightforward plotting of `NDData` objects with 1- and 2-dimensional datasets using `matplotlib`:

```
>>> from matplotlib import pyplot as plt
>>> plt.plot(ndd)
```

This works because the `matplotlib` plotting functions automatically convert their inputs using `numpy.array`.

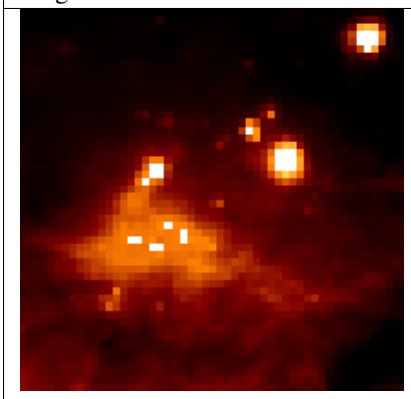
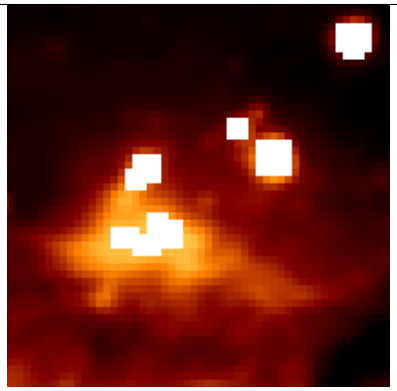
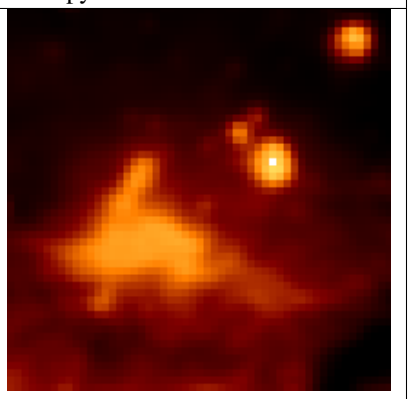
Convolution

Introduction

`astropy.nddata` includes a convolution function that offers improvements compared to the scipy `astropy.ndimage` convolution routines, including:

- Proper treatment of NaN values
- A single function for 1-D, 2-D, and 3-D convolution
- Improved options for the treatment of edges
- Both direct and Fast Fourier Transform (FFT) versions

The following thumbnails show the difference between Scipy's and Astropy's 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.

Original	Scipy convolve	Astropy convolve
		

Usage

Two convolution functions are provided. They are imported as:

```
from astropy.nddata 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 an 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 both 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 `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:

```
>>> convolve([1, 4, 5, 6, 5, 7, 8], [0.2, 0.6, 0.2])
array([ 0. ,  3.4,  5. ,  5.6,  5.6,  5.2,  0. ])
```

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:

```
>>> convolve([1, 4, 5, 6, np.nan, 7, 8], [0.2, 0.6, 0.2], boundary='extend')
array([ 1.6,  3.6,  5. ,  5.9,  6.5,  7.1,  7.8])
```

Kernels and arrays can be specified either as lists or as Numpy arrays. The following examples show how to construct a 1-d array as a list:

```
>>> kernel = [0, 1, 0]
>>> result = convolve(spectrum, kernel)
```

a 2-d array as a list:

```
>>> kernel = [[0, 1, 0], \
              [1, 2, 1], \
              [0, 1, 0]]
>>> result = convolve(image, kernel)
```

and a 3-d 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)
```

You can also use `make_kernel` to generate common n-dimensional kernels:

```
>>> make_kernel([3,3], 1, 'boxcar')
array([[ 0.  0.  0.]
       [ 0.  1.  0.]
       [ 0.  0.  0.]])
```

Subclassing `NDData` and `NDUncertainty`

Subclassing `NDUncertainty`

New error classes should sub-class from `NDUncertainty`, and should provide methods with the following API:

```
class MyUncertainty(NDUncertainty):

    def propagate_add(self, other_nddata, result_data):
        ...
        result_uncertainty = MyUncertainty(...)
        return result_uncertainty

    def propagate_subtract(self, other_nddata, result_data):
        ...
        result_uncertainty = MyUncertainty(...)
        return result_uncertainty

    def propagate_multiply(self, other_nddata, result_data):
        ...
        result_uncertainty = MyUncertainty(...)
        return result_uncertainty
```

```
    def propagate_divide(self, other_nddata, result_data):
        ...
        result_uncertainty = MyUncertainty(...)
        return result_uncertainty
```

All error sub-classes inherit an attribute `self.parent_nddata` that is automatically set to the parent `NDData` object that they are attached to. The arguments passed to the error propagation methods are `other_nddata`, which is the `NDData` object that is being combined with `self.parent_nddata`, and `result_data`, which is a Numpy array that contains the data array after the arithmetic operation. All these methods should return an error instance `result_uncertainty`, and should not modify `parent_nddata` directly. For subtraction and division, the order of the operations is `parent_nddata - other_nddata` and `parent_nddata / other_nddata`.

To make it easier and clearer to code up the error propagation, you can use variables with more explicit names, e.g:

```
class MyUncertainty(NDUncertainty):

    def propogate_add(self, other_nddata, result_data):

        left_uncertainty = self.parent.uncertainty.array
        right_uncertainty = other_nddata.uncertainty.array

        ...
```

Note that the above example assumes that the errors are stored in an `array` attribute, but this does not have to be the case.

For an example of a complete implementation, see `StdDevUncertainty`.

1.4.4 Reference/API

astropy.nddata Module

The `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

`convolve`(array, kernel[, boundary, ...])	Convolve an array with a kernel.
`convolve_fft`(array, kernel[, boundary, ...])	Convolve an ndarray with an nd-kernel.
`make_kernel`(kernelshape[, kernelwidth, ...])	Create a smoothing kernel for use with `convolve` or `convolve_fft`.

convolve

`astropy.nddata.convolution.convolve.convolve`(*array*, *kernel*, *boundary=None*, *fill_value=0.0*, *normalize_kernel=False*)

Convolve an array with a kernel.

This routine differs from `scipy.ndimage.filters.convolve` because it includes a special treatment for `NaN` values. Rather than including `NaN`s in the convolution calculation, which causes large `NaN` holes in the convolved image, `NaN` values are replaced with interpolated values using the kernel as an interpolation function.

Parameters

array : `numpy.ndarray`

The array to convolve. This should be a 1, 2, or 3-dimensional array or a list or a set of

nested lists representing a 1, 2, or 3-dimensional array.

kernel : numpy.ndarray

The convolution kernel. The number of dimensions should match those for the array, and the dimensions should be odd in all directions.

boundary : str, optional

A flag indicating how to handle boundaries:

- 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 `fill_value`.
- **'wrap'**
  Periodic boundary that wrap to the other side of `array`.
- **'extend'**
  Set values outside the array to the nearest `array` value.

fill_value : float, optional

The value to use outside the array when using boundary='fill'

normalize_kernel : bool, optional

Whether to normalize the kernel prior to convolving

Returns

result : numpy.ndarray

An array with the same dimensions and type as the input array, convolved with kernel.

Notes

Masked arrays are not supported at this time.

convolve_fft

astropy.nddata.convolution.convolve.convolve_fft(*array*, *kernel*, *boundary='fill'*, *fill_value=0*, *crop=True*, *return_fft=False*, *fft_pad=True*, *psf_pad=False*, *interpolate_nan=False*, *quiet=False*, *ignore_edge_zeros=False*, *min_wt=0.0*, *normalize_kernel=False*, *fft_type=None*, *nthreads=None*)

Convolve an ndarray with an nd-kernel. Returns a convolved image with shape = array.shape. Assumes kernel is centered.

convolve_fft differs from scipy.signal.fftconvolve in a few ways:

- can treat NaN's as zeros or interpolate over them
- defaults to using the faster FFTW algorithm if installed
- (optionally) pads to the nearest 2^n size to improve FFT speed
- only operates in mode='same' (i.e., the same shape array is returned) mode

Parameters

array : `numpy.ndarray`

Array to be convolved with `kernel`

kernel : `numpy.ndarray`

Will be normalized if `normalize_kernel` is set. Assumed to be centered (i.e., shifts may result if your kernel is asymmetric)

boundary : {'fill', 'wrap'}

A flag indicating how to handle boundaries:

- 'fill': set values outside the array boundary to fill_value (default)
- 'wrap': periodic boundary

interpolate_nan : bool

The convolution will be re-weighted assuming NAN values are meant to be ignored, not treated as zero. If this is off, all NaN values will be treated as zero.

ignore_edge_zeros : bool

Ignore the zero-pad-created zeros. This will effectively decrease the kernel area on the edges but will not re-normalize the kernel. This parameter may result in 'edge-brightening' effects if you're using a normalized kernel

min_wt : float

If ignoring NANs/zeros, force all grid points with a weight less than this value to NAN (the weight of a grid point with *no* ignored neighbors is 1.0). If `min_wt` == 0.0, then all zero-weight points will be set to zero instead of NAN (which they would be otherwise, because 1/0 = nan). See the examples below

normalize_kernel : function or boolean

If specified, this is the function to divide kernel by to normalize it. e.g., normalize_kernel=np.sum means that kernel will be modified to be: kernel = kernel / np.sum(kernel). If True, defaults to normalize_kernel = np.sum

Returns

default : ndarray

array convolved with `kernel`. If `return_fft` is set, returns fft(**array**) * fft(`kernel`). If crop is not set, returns the image, but with the fft-padded size instead of the input size

Other Parameters

fft_pad : bool

Default on. Zero-pad image to the nearest 2^n

psf_pad : bool

Default off. Zero-pad image to be at least the sum of the image sizes (in order to avoid edge-wrapping when smoothing)

crop : bool

Default on. Return an image of the size of the largest input image. If the images are asymmetric in opposite directions, will return the largest image in both directions. For example, if an input image has shape [100,3] but a kernel with shape [6,6] is used, the output will be [100,6].

return_fft : bool

Return the fft(image)*fft(kernel) instead of the convolution (which is ifft(fft(image)*fft(kernel))). Useful for making PSDs.

nthreads : int

if fftw3 is installed, can specify the number of threads to allow FFTs to use. Probably only helpful for large arrays

fft_type : [None, 'fftw', 'scipy', 'numpy']

Which FFT implementation to use. If not specified, defaults to the type specified in the FFT_TYPE ConfigurationItem

See Also:

`convolve`
Convolve is a non-fft version of this code.

Examples

```
>>> convolve_fft([1,0,3],[1,1,1])
array([ 1.,  4.,  3.])

>>> convolve_fft([1,np.nan,3],[1,1,1])
array([ 1.,  4.,  3.])

>>> convolve_fft([1,0,3],[0,1,0])
array([ 1.,  0.,  3.])

>>> convolve_fft([1,2,3],[1])
array([ 1.,  2.,  3.])

>>> convolve_fft([1,np.nan,3],[0,1,0], interpolate_nan=True)
array([ 1.,  0.,  3.])

>>> convolve_fft([1,np.nan,3],[0,1,0], interpolate_nan=True, min_wt=1e-8)
array([ 1.,  nan,  3.])

>>> convolve_fft([1,np.nan,3],[1,1,1], interpolate_nan=True)
array([ 1.,  4.,  3.])

>>> convolve_fft([1,np.nan,3],[1,1,1], interpolate_nan=True, normalize_kernel=True)
array([ 1.,  2.,  3.])
```

make_kernel

`astropy.nddata.convolution.make_kernel.make_kernel`(*kernelshape*, *kernelwidth=3*, *kernel-type='gaussian'*, *trapslope=None*, *nor-malize_kernel=<function sum at 0x3716578>*, *force_odd=False*)

Create a smoothing kernel for use with `convolve` or `convolve_fft`.

Parameters

kernelshape : n-tuple

A tuple (or list or array) defining the shape of the kernel. The length of kernelshape determines the dimensionality of the resulting kernel

kernelwidth : float

Width of kernel in pixels (see definitions under `kerneltype`)

kerneltype : {'gaussian', 'boxcar', 'tophat', 'brickwall', 'airy', 'trapezoid'}

Defines the type of kernel to be generated. The following types are available:

- **'gaussian'**
  Uses a gaussian kernel with sigma = `kernelwidth` (in pixels), i.e. kernel = exp(-r**2 / (2*sigma**2)) where r is the radius.
- **'boxcar'**
  A `kernelwidth` x `kernelwidth` square kernel, i.e., kernel = (x < `kernelwidth`) * (y < `kernelwidth`)
- **'tophat'**
  A flat circle with radius = `kernelwidth`, i.e., kernel = (r < `kernelwidth`)
- **'brickwall' or 'airy'**
  A kernel using the airy function from optics. It requires `scipy.special` for the bessel function. See e.g., http://en.wikipedia.org/wiki/Airy_disk.
- **'trapezoid'**
  A kernel like 'tophat' but with sloped edges. It is effectively a cone chopped off at the `kernelwidth` radius.

trapslope : float

Slope of the trapezoid kernel. Only used if `kerneltype` == 'trapezoid'

normalize_kernel : function

Function to use for kernel normalization

force_odd : boolean

If set, forces the kernel to have odd dimensions (needed for convolve w/o ffts)

Returns

kernel : ndarray

An N-dimensional float array

Examples

```
>>> make_kernel([3,3],1,'boxcar')
array([[ 0.  0.  0.]
       [ 0.  1.  0.]
       [ 0.  0.  0.]])

>>> make_kernel([9],1) # Gaussian by default
array([  1.33830625e-04   4.43186162e-03   5.39911274e-02   2.41971446e-01
         3.98943469e-01   2.41971446e-01   5.39911274e-02   4.43186162e-03
         1.33830625e-04])

>>> make_kernel([3,3],3,'boxcar')
array([[ 0.11111111,  0.11111111,  0.11111111],
```

```
       [ 0.11111111,  0.11111111,  0.11111111],
       [ 0.11111111,  0.11111111,  0.11111111]])

>>> make_kernel([3,3],1.4,'tophat')
array([[ 0. ,  0.2,  0. ],
       [ 0.2,  0.2,  0.2],
       [ 0. ,  0.2,  0. ]])
```

Classes

FlagCollection(*args, **kwargs)	The purpose of this class is to provide a dictionary for containing arrays of flags for the
IncompatibleUncertaintiesException	This exception should be used to indicate cases in which uncertainties with two differen
MissingDataAssociationException	This exception should be used to indicate that an uncertainty instance has not been assoc
NDData(data[, uncertainty, mask, flags, ...])	A Superclass for array-based data in Astropy.
NDUncertainty	This is the base class for uncertainty classes used with NDData.
StdDevUncertainty([array, copy])	A class for standard deviation uncertaintys

FlagCollection

class astropy.nddata.flag_collection.FlagCollection(**args*, ***kwargs*)

Bases: collections.OrderedDict

The purpose of this class is to provide a dictionary for containing arrays of flags for the NDData class. Flags should be stored in Numpy arrays that have the same dimensions as the parent data, so the FlagCollection class adds shape checking to an ordered dictionary class.

The FlagCollection should be initialized like an OrderedDict, but with the addition of a shape= keyword argument used to pass the NDData shape.

IncompatibleUncertaintiesException

exception astropy.nddata.nduncertainty.IncompatibleUncertaintiesException

This exception should be used to indicate cases in which uncertainties with two different classes can not be propagated.

MissingDataAssociationException

exception astropy.nddata.nduncertainty.MissingDataAssociationException

This exception should be used to indicate that an uncertainty instance has not been associated with a parent NDData object.

NDData

class astropy.nddata.nddata.NDData(*data*, *uncertainty=None*, *mask=None*, *flags=None*, *wcs=None*, *meta=None*, *units=None*, *copy=True*)

Bases: object

A Superclass for array-based data in Astropy.

The key distinction from raw numpy arrays is the presence of additional metadata such as uncertainties, a mask, units, flags, and/or a coordinate system.

Parameters

data : ndarray or NDData

The actual data contained in this NDData object.

uncertainty : `NDUncertainty`, optional

Uncertainties on the data.

mask : `ndarray`, optional

Mask for the data, given as a boolean Numpy array with a shape matching that of the data. The values must be `False` where the data is *valid* and `True` when it is not (as for Numpy masked arrays).

flags : `ndarray` or `FlagCollection`, optional

Flags giving information about each pixel. These can be specified either as a Numpy array of any type with a shape matching that of the data, or as a `FlagCollection` instance which has a shape matching that of the data.

wcs : undefined, optional

WCS-object containing the world coordinate system for the data.

> **Warning:** This is not yet defind because the discussion of how best to represent this class's WCS system generically is still under consideration. For now just leave it as None

meta : `dict`-like object, optional

Metadata for this object. "Metadata" here means all information that is included with this object but not part of any other attribute of this particular object. e.g., creation date, unique identifier, simulation parameters, exposure time, telescope name, etc.

units : `astropy.units.UnitBase` instance or str, optional

The units of the data.

copy : bool, optional

If True, the array will be *copied* from the provided `data`, otherwise it will be referenced if possible (see `numpy.array` `copy` argument for details).

Raises

ValueError :

If the `uncertainty` or `mask` inputs cannot be broadcast (e.g., match shape) onto `data`.

Notes

`NDData` objects can be easily converted to a regular Numpy array using `numpy.asarray`

For example:

```
>>> from astropy.nddata import NDData
>>> import numpy as np
>>> x = NDData([1,2,3])
>>> np.asarray(x)
array([1, 2, 3])
```

If the `NDData` object has a `mask`, `numpy.asarray` will return a Numpy masked array.

This is useful, for example, when plotting a 2D image using matplotlib:

```
>>> from astropy.nddata import NDData
>>> from matplotlib import pyplot as plt
>>> x = NDData([[1,2,3], [4,5,6]])
>>> plt.imshow(x)
```

Attributes Summary

ndim	integer dimensions of this object's data
dtype	numpy.dtype of this object's data.
units	
shape	shape tuple of this object's data.
meta	
size	integer size of this object's data.
uncertainty	
mask	
flags	

Methods Summary

divide(operand[, propagate_uncertainties])	Divide another dataset (operand) to this dataset.
convert_units_to(unit[, equivalencies])	Returns a new NDData object whose values have been converted to a new unit.
multiply(operand[, propagate_uncertainties])	Multiply another dataset (operand) to this dataset.
subtract(operand[, propagate_uncertainties])	Subtract another dataset (operand) to this dataset.
add(operand[, propagate_uncertainties])	Add another dataset (operand) to this dataset.

Attributes Documentation

ndim
: integer dimensions of this object's data

dtype
: numpy.dtype of this object's data.

units

shape
: shape tuple of this object's data.

meta

size
: integer size of this object's data.

uncertainty

mask

flags

Methods Documentation

divide(*operand*, *propagate_uncertainties=True*)
Divide another dataset (operand) to this dataset.

Parameters
operand : NDData

The second operand in the operation a / b

propagate_uncertainties : bool

Whether to propagate uncertainties following the propagation rules defined by the class used for the uncertainty attribute.

Returns
result : NDData

The resulting dataset

Notes

This method requires the datasets to have identical WCS properties, equivalent units, and identical shapes. Flags and meta-data get set to None in the resulting dataset. The unit in the result is the same as the unit in self. Uncertainties are propagated, although correlated errors are not supported by any of the built-in uncertainty classes. If uncertainties are assumed to be correlated, a warning is issued by default (though this can be disabled via the WARN_UNSUPPORTED_CORRELATED configuration item). Values masked in either dataset before the operation are masked in the resulting dataset.

convert_units_to(*unit*, *equivalencies=[]*)
Returns a new NDData object whose values have been converted to a new unit.

Parameters
unit : astropy.units.UnitBase instance or str

The unit to convert to.

equivalencies : list of equivalence pairs, optional

A list of equivalence pairs to try if the units are not directly convertible. See *Equivalencies*.

Returns
result : NDData

The resulting dataset

Raises
UnitsException :

If units are inconsistent.

multiply(*operand*, *propagate_uncertainties=True*)
Multiply another dataset (operand) to this dataset.

Parameters
operand : NDData

The second operand in the operation a * b

propagate_uncertainties : bool

Whether to propagate uncertainties following the propagation rules defined by the class used for the `uncertainty` attribute.

Returns

result : `NDData`

The resulting dataset

Notes

This method requires the datasets to have identical WCS properties, equivalent units, and identical shapes. Flags and meta-data get set to None in the resulting dataset. The unit in the result is the same as the unit in `self`. Uncertainties are propagated, although correlated errors are not supported by any of the built-in uncertainty classes. If uncertainties are assumed to be correlated, a warning is issued by default (though this can be disabled via the `WARN_UNSUPPORTED_CORRELATED` configuration item). Values masked in either dataset before the operation are masked in the resulting dataset.

`subtract`(*operand*, *propagate_uncertainties=True*)

Subtract another dataset (`operand`) to this dataset.

Parameters

operand : `NDData`

The second operand in the operation a - b

propagate_uncertainties : bool

Whether to propagate uncertainties following the propagation rules defined by the class used for the `uncertainty` attribute.

Returns

result : `NDData`

The resulting dataset

Notes

This method requires the datasets to have identical WCS properties, equivalent units, and identical shapes. Flags and meta-data get set to None in the resulting dataset. The unit in the result is the same as the unit in `self`. Uncertainties are propagated, although correlated errors are not supported by any of the built-in uncertainty classes. If uncertainties are assumed to be correlated, a warning is issued by default (though this can be disabled via the `WARN_UNSUPPORTED_CORRELATED` configuration item). Values masked in either dataset before the operation are masked in the resulting dataset.

`add`(*operand*, *propagate_uncertainties=True*)

Add another dataset (`operand`) to this dataset.

Parameters

operand : `NDData`

The second operand in the operation a + b

propagate_uncertainties : bool

Whether to propagate uncertainties following the propagation rules defined by the class used for the `uncertainty` attribute.

Returns

result : `NDData`

The resulting dataset

Notes

This method requires the datasets to have identical WCS properties, equivalent units, and identical shapes. Flags and meta-data get set to None in the resulting dataset. The unit in the result is the same as the unit in `self`. Uncertainties are propagated, although correlated errors are not supported by any of the built-in uncertainty classes. If uncertainties are assumed to be correlated, a warning is issued by default (though this can be disabled via the `WARN_UNSUPPORTED_CORRELATED` configuration item). Values masked in either dataset before the operation are masked in the resulting dataset.

NDUncertainty

class `astropy.nddata.nduncertainty.NDUncertainty`

Bases: `object`

This is the base class for uncertainty classes used with NDData. It is implemented as an abstract class and should never be directly instantiated.

Classes inheriting from NDData should overload the `propagate_*` methods, keeping the call signature the same. The propagate methods can assume that a `parent_nddata` attribute is present which links to the parent_nddata dataset, and take an `NDData` instance as the positional argument, *not* an `NDUncertainty` instance, because the `NDData` instance can be used to access both the data and the uncertainties (some propagations require the data values).

Attributes Summary

`supports_correlated`	bool(x) -> bool
`parent_nddata`	

Methods Summary

`propagate_subtract`(other_nddata, result_data)	Propagate uncertainties for subtraction.
`propagate_add`(other_nddata, result_data)	Propagate uncertainties for addition.
`propagate_multiply`(other_nddata, result_data)	Propagate uncertainties for multiplication.
`propagate_divide`(other_nddata, result_data)	Propagate uncertainties for division.

Attributes Documentation

`supports_correlated` = **False**

`parent_nddata`

Methods Documentation

`propagate_subtract`(*other_nddata*, *result_data*)

Propagate uncertainties for subtraction.

Parameters
other_nddata : NDData instance

The data for the second other_nddata in a + b

result_data : `ndarray` instance

The data array that is the result of the addition

Returns
result_uncertainty : NDUncertainty instance

The resulting uncertainty

Raises
IncompatibleUncertaintiesException :

Raised if the method does not know how to add the uncertainties

`propagate_add`(*other_nddata*, *result_data*)
Propagate uncertainties for addition.

Parameters
other_nddata : NDData instance

The data for the second other_nddata in a + b

result_data : `ndarray` instance

The data array that is the result of the addition

Returns
result_uncertainty : NDUncertainty instance

The resulting uncertainty

Raises
IncompatibleUncertaintiesException :

Raised if the method does not know how to add the uncertainties

`propagate_multiply`(*other_nddata*, *result_data*)
Propagate uncertainties for multiplication.

Parameters
other_nddata : NDData instance

The data for the second other_nddata in a + b

result_data : `ndarray` instance

The data array that is the result of the addition

Returns
result_uncertainty : NDUncertainty instance

The resulting uncertainty

`propagate_divide`(*other_nddata*, *result_data*)
Propagate uncertainties for division.

Parameters
other_nddata : NDData instance

The data for the second other_nddata in a + b

result_data : `ndarray` instance

The data array that is the result of the addition

Returns

result_uncertainty : NDUncertainty instance

The resulting uncertainty

StdDevUncertainty

class astropy.nddata.nduncertainty.StdDevUncertainty(*array=None*, *copy=True*)

Bases: astropy.nddata.nduncertainty.NDUncertainty

A class for standard deviation uncertaintys

Attributes Summary

support_correlated	bool(x) -> bool
parent_nddata	
array	

Methods Summary

propagate_add(other_nddata, result_data)	Propagate uncertainties for addition.
propagate_subtract(other_nddata, result_data)	Propagate uncertainties for subtraction.
propagate_multiply(other_nddata, result_data)	Propagate uncertainties for mutliplication.
propagate_divide(other_nddata, result_data)	Propagate uncertainties for division.

Attributes Documentation

support_correlated = **False**

parent_nddata

array

Methods Documentation

propagate_add(*other_nddata*, *result_data*)

Propagate uncertainties for addition.

Parameters

other_nddata : NDData instance

The data for the second other_nddata in a + b

result_data : ndarray instance

The data array that is the result of the addition

Returns

result_uncertainty : NDUncertainty instance

The resulting uncertainty

Raises

IncompatibleUncertaintiesException :

Raised if the method does not know how to propagate the uncertainties

`propagate_subtract`(*other_nddata*, *result_data*)

Propagate uncertainties for subtraction.

Parameters

other_nddata : NDData instance

The data for the second other_nddata in a + b

result_data : `ndarray` instance

The data array that is the result of the addition

Returns

result_uncertainty : NDUncertainty instance

The resulting uncertainty

Raises

IncompatibleUncertaintiesException :

Raised if the method does not know how to propagate the uncertainties

`propagate_multiply`(*other_nddata*, *result_data*)

Propagate uncertainties for mutliplication.

Parameters

other_nddata : NDData instance

The data for the second other_nddata in a + b

result_data : `ndarray` instance

The data array that is the result of the addition

Returns

result_uncertainty : NDUncertainty instance

The resulting uncertainty

Raises

IncompatibleUncertaintiesException :

Raised if the method does not know how to propagate the uncertainties

`propagate_divide`(*other_nddata*, *result_data*)

Propagate uncertainties for division.

Parameters

other_nddata : NDData instance

The data for the second other_nddata in a + b

result_data : `ndarray` instance

The data array that is the result of the addition

Returns

result_uncertainty : NDUncertainty instance

The resulting uncertainty

Raises

IncompatibleUncertaintiesException :

Raised if the method does not know how to propagate the uncertainties

1.5 Units (`astropy.units`)

1.5.1 Introduction

`astropy.units` is a Python package to handle defining and converting between physical units, and performing arithmetic with physical quantities (numbers with associated units).

1.5.2 Getting Started

```
>>> from astropy import units as u
>>> # Convert from parsec to meter
>>> u.pc.to(u.m)
3.0856776e+16
>>> cms = u.cm / u.s
>>> mph = u.mile / u.hour
>>> cms.to(mph, 1)
0.02236936292054402
>>> cms.to(mph, [1., 1000., 5000.])
array([  2.23693629e-02,   2.23693629e+01,   1.11846815e+02])
```

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

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

`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:

```
# Wavelength to frequency doesn't normally work
>>> u.nm.to(u.Hz, [1000, 2000])
UnitsException: 'nm' (length) and 'Hz' (frequency) are not convertible
# ...but by passing an equivalency unit (spectral()), it does...
>>> u.nm.to(u.Hz, [1000, 2000], equivs=u.spectral())
array([  2.99792458e+14,   1.49896229e+14])
>>> u.nm.to(u.eV, [1000, 2000], equivs=u.spectral())
array([ 1.23984201,  0.61992101])
```

Also included in the `astropy.units` package is the `Quantity` object, which represents a numerical value with an associated unit. These objects support arithmetic with other numbers and `Quantity` objects and preserve units:

```
>>> from astropy import units as u
>>> 15.1*u.meter / (32.0*u.second)
<Quantity 0.471875 m / (s)>
>>> 3.0*u.kilometer / (130.51*u.meter/u.second)
<Quantity 0.0229867443108 km s / (m)>
>>> (3.0*u.kilometer / (130.51*u.meter/u.second)).simplify_units()
<Quantity 22.9867443108 s>
```

1.5.3 Using `astropy.units`

Standard units

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

All units are defined in term 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, one does not need to worry about the various kinds of unit classes unless one wants to design a more complex case.

There are many units already predefined in the module. One may use the following function to list all 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          | 9.109383e-31 kg |            ,
  M_p          | 1.672622e-27 kg |            ,
  g            | 1.000000e-03 kg | gram       ,
  kg           | irreducible     | kilogram   ,
  lb           | 4.535924e-01 kg | pound      ,
  oz           | 2.834952e-02 kg | ounce      ,
  solMass      | 1.989100e+30 kg |            ,
  t            | 1.000000e+03 kg | tonne      ,
  ton          | 9.071847e+02 kg |            ,
  u            | 1.660539e-27 kg | Da, Dalton ,
]
```

The dimensionless unit

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

To obtain the dimensionless unit, use the `dimensionless` object:

```
>>> from astropy import units as u
>>> u.dimensionless
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. For example:

```
>>> 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` in Python code is preferred for readability:

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

Composing and defining units

Units can be composed together using the regular Python numeric operators. For example:

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

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

```
>>> 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.

Creating a new fundamental unit is simple:

```
>>> 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)
>>> rofl.to(titter, 1)
240
```

Reducing a unit to its irreducible parts

A unit can be decomposed into its irreducible parts using the `decompose` method:

```
>>> u.Ry
Unit("Ry")
>>> u.Ry.decompose()
Unit("2.18e-18 m2 kg / (s2)")
```

Unit Conversion

There are two ways of handling conversions between units.

Direct Conversion

In this case, given a source and destination unit, the value(s) in the new units is(are) returned.

```
>>> from astropy import units as u
>>> u.pc.to(u.m, 3.26)
1.0059317615e+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.])
```

Obtaining a Conversion Function

Finally, one may obtain a function that can be used to convert to the new unit. Normally this may seem like overkill when all one needs to do is multiply by a scale factor, but there are cases when the transformation between units may not be as simple as a single scale factor, for example when a custom equivalency table is in use.

Conversion to different units involves obtaining a conversion function and then applying it to the value, or values to be converted.

```
>>> cms = u.cm / u.s
>>> cms_to_mph = cms.get_converter(u.mile / u.hour)
>>> cms_to_mph(100.)
2.2366936292054402
>>> cms_to_mph([1000, 2000])
array([ 22.36936292,  44.73872584])
```

Incompatible Conversions

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

```
>>> cms.to(u.mile)
...
UnitsException: 'cm / (s)' (speed) and 'mi' (length) are not convertible
```

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

```
>>> u.m.is_equivalent(u.foot)
True
>>> u.m.is_equivalent("second")
False
>>> (u.m ** 2).is_equivalent(u.acre)
True
```

Unit formats

Units can be created from strings using the `Unit` class:

```
>>> from astropy import units as u
>>> u.Unit("m")
Unit("m")
>>> u.Unit("erg / (s cm2)")
Unit("erg / (s cm2)")
```

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')`).

Units can be converted to strings using the `to_string` method:

```
>>> fluxunit = u.erg / (u.cm ** 2 * u.s)
>>> fluxunit.to_string()
u'erg / (cm2 s)'
```

By default, the string format used is referred to as the "generic" format, which is based on syntax of the FITS standard's format for representing units, but supports all of the units defined within the `astropy.units` framework, including user-defined units. The `Unit` and `to_string` functions also take an optional `format` parameter to select a different format.

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 proposed IVOA 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, for example, by VOTable versions 1.2 and earlier.

`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
  ```

$$\frac{\mathrm{erg}}{\mathrm{s\ cm^2}}$$

- `"console"`: Writes a multi-line 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
2.18×10⁻¹⁸ ─────
            s²
```

Unrecognized Units

Since many files in 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:

```
>>> u.Unit("m/s/s")  # The FITS standard only allows one '/'
ValueError: Expected end of text (at char 3) in 'm/s/s'
```

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.

So, for example, one can do:

```
>>> x = u.Unit("m/s/s", parse_strict="warn")
WARNING: UnitsWarning: 'm/s/s' did not parse using format 'generic'.
Expected end of text (at char 3) in 'm/s/s' [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()
'm/s/s'
>>> x.to(u.km / u.s / u.s)
ValueError: The unit 'm/s/s' is unrecognized.  It can not be converted to
other units.
>>> x / u.m
ValueError: The unit 'm/s/s' 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. This will describe how to use two of the equivalencies include in `astropy.units` and then describe how to define new equivalencies.

Equivalencies are used by passing a list of equivalency pairs to the `equivs` keyword argument of `to` or `get_converter` methods.

Built-in equivalencies

Spectral Units `spectral` is a function that returns an equivalency list to handle conversions between wavelength, frequency and energy.

Length and frequency are not normally convertible, so `to` raises an exception:

```
>>> from astropy import units as u
>>> u.nm.to(u.Hz, [1000, 2000])
UnitsException: 'nm' (length) and 'Hz' (frequency) are not convertible
```

However, when passing the result of `spectral` as the third argument to the `to` method, wavelength, frequency and energy can be converted.

```
>>> u.nm.to(u.Hz, [1000, 2000], equivs=u.spectral())
array([  2.99792458e+14,   1.49896229e+14])
>>> u.nm.to(u.eV, [1000, 2000], equivs=u.spectral())
array([ 1.23984201,  0.61992101])
```

These equivalencies even work with non-base units:

```
>>> # Inches to calories
>>> u.inch.to(u.Cal, 1, equivs=u.spectral())
1.869180759162485e-27
```

Spectral Flux Density Units There is also support for spectral flux density 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 unit and value for the spectral location. For example:

```
>>> u.Jy.to(u.erg / u.cm**2 / u.s / u.Hz, 1., equivs=u.spectral_density(u.AA, 3500))
1.0000000000000001e-23
```

```
>>> u.Jy.to(u.erg / u.cm**2 / u.s / u.micron, 1., equivs=u.spectral_density(u.AA, 3500))
2.4472853714285712e-08
```

Writing new equivalencies

An equivalence list is just a list of tuples, where each tuple has 4 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.

For example, 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, equivs=liters_water)
1.0
```

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

```
>>> u.gallon.to(u.pound, 1, equivs=liters_water)
8.345404463333525
```

And it also works in the other direction:

```
>>> u.lb.to(u.pint, 1, equivs=liters_water)
0.9586114172355458
```

Displaying available equivalencies

The `find_equivalent_units` function also understands equivalencies. For example, without passing equivalencies, there are no compatible units for `Hz` in the standard set:

```
>>> u.Hz.find_equivalent_units()
  Primary name | Unit definition | Aliases
[
  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(equivs=u.spectral())
  Primary name | Unit definition           | Aliases
[
  AU           | 1.495979e+11 m            | au                                ,
  Angstrom     | 1.000000e-10 m            | AA, angstrom                      ,
  BTU          | 1.055056e+03 kg m2 / (s2) | btu                               ,
  Hz           | 1 / (s)                   | Hertz, hertz                      ,
  J            | kg m2 / (s2)              | Joule, joule                      ,
  Ry           | 2.179872e-18 kg m2 / (s2) | rydberg                           ,
  cal          | 4.184000e+00 kg m2 / (s2) | calorie                           ,
  eV           | 1.602177e-19 kg m2 / (s2) | electronvolt                      ,
  erg          | 1.000000e-07 kg m2 / (s2) |                                   ,
  ft           | 3.048000e-01 m            | foot                              ,
  inch         | 2.540000e-02 m            |                                   ,
  kcal         | 4.184000e+03 kg m2 / (s2) | Cal, Calorie, kilocal, kilocalorie ,
```

```
  lyr          | 9.460730e+15 m           |                                ,
  m            | irreducible              | meter                          ,
  mi           | 1.609344e+03 m           | mile                           ,
  micron       | 1.000000e-06 m           |                                ,
  pc           | 3.085678e+16 m           | parsec                         ,
  solRad       | 6.955080e+08 m           |                                ,
  yd           | 9.144000e-01 m           | yard                           ,
]
```

Quantity

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

Creating Quantity instances

`Quantity` objects are created through multiplication or divison with `Unit` objects. For example, to create a `Quantity` to represent $15\frac{m}{s}$

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

or $1.14s^{-1}$

```
>>> 1.14/u.s
<Quantity 1.14 1 / (s)>
```

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)>
```

Arithmetic

Addition and Subtraction Addition or subtraction between `Quantity` objects is supported when their units are equivalent (see `Unit` documentation). 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 is not supported between `Quantity` objects and basic numeric types

```
>>> 13.5*u.km + 19.412
TypeError: Object of type '<type 'float'>' cannot be added with a Quantity object. Addition is only supported between Qua
```

Multiplication and Division Multiplication and division is 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**

```
>>> 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 20.0 cm / (m)>
```

For multiplication, you can choose 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 15433.0 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 `simplify_units()` method

```
>>> (20.*u.cm / (1.*u.m)).simplify_units()
<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 0.341950972779 cm kg m / (ms s)>
>>> (15.*u.kg * 32.*u.cm * 15*u.m / (11.*u.s * 1914.15*u.ms)).simplify_units()
<Quantity 3.41950972779 kg m2 / (s2)>
```

1.5.4 See Also

- FITS Standard for units in FITS.
- The proposed IVOA 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

1.5.5 Reference/API

astropy.units.core Module

Core units classes and functions

Functions

def_unit(s[, represents, register, doc, ...])	Factory function for defining new units.

def_unit

astropy.units.core.def_unit(*s*, *represents=None*, *register=None*, *doc=None*, *format=None*, *prefixes=False*, *exclude_prefixes=[]*)

Factory function for defining new units.

Parameters

names : str or list of str

The name of the unit. If a list, the first element is the canonical (short) name, and the rest of the elements are aliases.

represents : UnitBase instance, optional

The unit that this named unit represents. If not provided, a new `IrreducibleUnit` is created.

register : boolean, optional

When `True`, also register the unit in the standard unit namespace. Default is `False`.

doc : str, optional

A docstring describing the unit.

format : dict, optional

A mapping to format-specific representations of this unit. For example, for the `Ohm` unit, it might be nice to have it displayed as `\Omega` by the `latex` formatter. In that case, `format` argument should be set to:

```
{'latex': r'\Omega'}
```

prefixes : bool, optional

When `True`, generate all of the SI prefixed versions of the unit as well. For example, for a given unit `m`, will generate `mm`, `cm`, `km`, etc. Default is `False`. This function always returns the base unit object, even if multiple scaled versions of the unit were created.

exclude_prefixes : list of str, optional

If any of the SI prefixes need to be excluded, they may be listed here. For example, `Pa` can be interpreted either as "petaannum" or "Pascal". Therefore, when defining the prefixes for `a`, `exclude_prefixes` should be set to `["P"]`.

Returns

unit : UnitBase object

The newly-defined unit, or a matching unit that was already defined.

Classes

UnitsException	The base class for unit-specific exceptions.
UnitsWarning	The base class for unit-specific exceptions.
UnitBase	Abstract base class for units.
NamedUnit(st[, register, doc, format])	The base class of units that have a name.
IrreducibleUnit(st[, register, doc, format])	Irreducible units are the units that all other units are defined in terms of.
Unit(st[, represents, register, doc, format])	The main unit class.
CompositeUnit(scale, bases, powers)	Create a composite unit using expressions of previously defined units.
PrefixUnit(st[, represents, register, doc, ...])	A unit that is simply a SI-prefixed version of another unit.
UnrecognizedUnit(st)	A unit that did not parse correctly.

UnitsException

exception astropy.units.core.UnitsException

The base class for unit-specific exceptions.

UnitsWarning

exception astropy.units.core.UnitsWarning

The base class for unit-specific exceptions.

UnitBase

class astropy.units.core.UnitBase

Bases: object

Abstract base class for units.

Most of the arithmetic operations on units are defined in this base class.

Should not be instantiated by users directly.

Attributes Summary

physical_type	Return the physical type on the unit.

Methods Summary

is_dimensionless()	Returns True if this unit translates into a scalar quantity without a unit.
get_converter(other[, equivalencies])	Return the conversion function to convert values from self to the specified unit.
to(other[, value, equivalencies])	Return the converted values in the specified unit.
simplify()	Compresses a possibly composite unit down to a single instance.
is_equivalent(other[, equivalencies])	Returns True if this unit is equivalent to other.
find_equivalent_units([equivalencies])	Return a list of all the units that are the same type as the specified unit.
to_string([format])	Output the unit in the given format as a string.

Continued on next page

Table 1.12 – continued from previous page

decompose()	Return a unit object composed of only irreducible units.
in_units(other[, value, equivalencies])	Alias for to for backward compatibility with pynbody.

Attributes Documentation

physical_type
: Return the physical type on the unit.

Examples

```
>>> u.m.physical_type
'length'
```

Methods Documentation

is_dimensionless()
: Returns True if this unit translates into a scalar quantity without a unit.

Examples

```
>>> ((2 * u.m) / (3 * u.m)).is_dimensionless()
True
>>> (2 * u.m).is_dimensionless()
False
```

get_converter(*other*, *equivalencies*=[ ])
: Return the conversion function to convert values from self to the specified unit.

Parameters
other : unit object or string

The unit to convert to.

equivalencies : list of equivalence pairs, optional

A list of equivalence pairs to try if the units are not directly convertible. See *Equivalencies*.

Returns
func : callable

A callable that normally expects a single argument that is a scalar value or an array of values (or anything that may be converted to an array).

Raises
UnitsException :

If units are inconsistent

to(*other*, *value=1.0*, *equivalencies*=[ ])
: Return the converted values in the specified unit.

Parameters
other : unit object or string

The unit to convert to.

value : scalar int or float, or sequence that can be converted to array, optional

Value(s) in the current unit to be converted to the specified unit. If not provided, defaults to 1.0

equivalencies : list of equivalence pairs, optional

A list of equivalence pairs to try if the units are not directly convertible. See *Equivalencies*.

Returns

values : scalar or array

Converted value(s). Input value sequences are returned as numpy arrays.

Raises

UnitException :

If units are inconsistent

`simplify()`

Compresses a possibly composite unit down to a single instance.

`is_equivalent(`*other*, *equivalencies*=[ ]`)`

Returns `True` if this unit is equivalent to `other`.

Parameters

other : unit object or string

The unit to convert to.

equivalencies : list of equivalence pairs, optional

A list of equivalence pairs to try if the units are not directly convertible. See *Equivalencies*.

Returns

bool :

`find_equivalent_units(`*equivalencies*=[ ]`)`

Return a list of all the units that are the same type as the specified unit.

Parameters

u : Unit instance or string

The `Unit` to find similar units to.

equivalencies : list of equivalence pairs, optional

A list of equivalence pairs to also list. See *Equivalencies*.

Returns

units : list of `UnitBase`

A list of unit objects that match `u`. A subclass of `list` (`EquivalentUnitsList`) is returned that pretty-prints the list of units when output.

`to_string(`*format=u'generic'*`)`

Output the unit in the given format as a string.

Parameters

format : `astropy.format.Base` instance or str

The name of a format or a formatter object. If not provided, defaults to the generic format.

`decompose()`
Return a unit object composed of only irreducible units.

Parameters
None :

Returns
unit : CompositeUnit object

New object containing only irreducible unit objects.

`in_units(`*other*, *value=1.0*, *equivalencies=*`[])`
Alias for `to` for backward compatibility with pynbody.

NamedUnit
class `astropy.units.core.NamedUnit(`*st*, *register=False*, *doc=None*, *format=None*`)`
Bases: `astropy.units.core.UnitBase`

The base class of units that have a name.

Parameters
st : str or list of str

The name of the unit. If a list, the first element is the canonical (short) name, and the rest of the elements are aliases.

register : boolean, optional

When `True`, also register the unit in the standard unit namespace. Default is `False`.

doc : str, optional

A docstring describing the unit.

format : dict, optional

A mapping to format-specific representations of this unit. For example, for the `Ohm` unit, it might be nice to have it displayed as `\Omega` by the `latex` formatter. In that case, `format` argument should be set to:

```
{'latex': r'\Omega'}
```

Raises
ValueError :

If any of the given unit names are already in the registry.

ValueError :

If any of the given unit names are not valid Python tokens.

Attributes Summary

`name`	Returns the canonical (short) name associated with this unit.
`names`	Returns all of the names associated with this unit.
`aliases`	Returns the alias (long) names for this unit.

Methods Summary

get_format_name(format)	Get a name for this unit that is specific to a particular format.

Attributes Documentation

name
: Returns the canonical (short) name associated with this unit.

names
: Returns all of the names associated with this unit.

aliases
: Returns the alias (long) names for this unit.

Methods Documentation

get_format_name(*format*)
: Get a name for this unit that is specific to a particular format.

  Uses the dictionary passed into the format kwarg in the constructor.

  Parameters
  format : str

  The name of the format

  Returns
  name : str

  The name of the unit for the given format.

IrreducibleUnit

class astropy.units.core.IrreducibleUnit(*st*, *register=False*, *doc=None*, *format=None*)
: Bases: astropy.units.core.NamedUnit

  Irreducible units are the units that all other units are defined in terms of.

  Examples are meters, seconds, kilograms, amperes, etc. There is only once instance of such a unit per type.

Methods Summary

decompose()	Return a unit object composed of only irreducible units.

Methods Documentation

decompose()
: Return a unit object composed of only irreducible units.

  Parameters
  None :

  Returns

unit : CompositeUnit object

New object containing only irreducible unit objects.

Unit

class astropy.units.core.Unit(*st*, *represents=None*, *register=False*, *doc=None*, *format=None*)

Bases: astropy.units.core.NamedUnit

The main unit class.

There are a number of different ways to construct a Unit, but always returns a UnitBase instance. If the arguments refer to an already-existing unit, that existing unit instance is returned, rather than a new one.

- From a string:

```
Unit(s, format=None, parse_strict='silent')
```

Construct from a string representing a (possibly compound) unit.

The optional format keyword argument specifies the format the string is in, by default "generic". For a description of the available formats, see astropy.units.format.

The optional parse_strict keyword controls what happens when an unrecognized unit string is passed in. It may be one of the following:

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

- From a number:

```
Unit(number)
```

Creates a dimensionless unit.

- From a UnitBase instance:

```
Unit(unit)
```

Returns the given unit unchanged.

- From None:

```
Unit()
```

Returns the null unit.

- The last form, which creates a new Unit is described in detail below.

Parameters

st : str or list of str

The name of the unit. If a list, the first element is the canonical (short) name, and the rest of the elements are aliases.

represents : UnitBase instance

The unit that this named unit represents.

register : boolean, optional

When True, also register the unit in the standard unit namespace. Default is False.

doc : str, optional

A docstring describing the unit.

format : dict, optional

A mapping to format-specific representations of this unit. For example, for the `Ohm` unit, it might be nice to have it displayed as `\Omega` by the `latex` formatter. In that case, `format` argument should be set to:

```
{'latex': r'\Omega'}
```

Raises

ValueError :

If any of the given unit names are already in the registry.

ValueError :

If any of the given unit names are not valid Python tokens.

Methods Summary

`decompose`()	Return a unit object composed of only irreducible units.

Methods Documentation

`decompose()`

Return a unit object composed of only irreducible units.

Parameters

None :

Returns

unit : CompositeUnit object

New object containing only irreducible unit objects.

CompositeUnit

class `astropy.units.core.CompositeUnit`(*scale*, *bases*, *powers*)

Bases: `astropy.units.core.UnitBase`

Create a composite unit using expressions of previously defined units.

Direct use of this class is not recommended. Instead use the factory function `Unit(...)` and arithmetic operators to compose units.

Parameters

scale : number

A scaling factor for the unit.

bases : sequence of `UnitBase`

A sequence of units this unit is composed of.

powers : sequence of numbers

A sequence of powers (in parallel with `bases`) for each of the base units.

Attributes Summary

bases	Return the bases of the composite unit.
scale	Return the scale of the composite unit.
powers	Return the powers of the composite unit.

Methods Summary

is_dimensionless()	Returns True if this unit translates into a scalar quantity without a unit.
simplify()	Compresses a possibly composite unit down to a single instance.
dimensionless_constant()	If this unit is dimensionless, return its scalar quantity.
decompose()	Return a unit object composed of only irreducible units.

Attributes Documentation

bases
: Return the bases of the composite unit.

scale
: Return the scale of the composite unit.

powers
: Return the powers of the composite unit.

Methods Documentation

is_dimensionless()
: Returns True if this unit translates into a scalar quantity without a unit.

Examples

```
>>> ((2 * u.m) / (3 * u.m)).is_dimensionless()
True
>>> (2 * u.m).is_dimensionless()
False
```

simplify()
: Compresses a possibly composite unit down to a single instance.

dimensionless_constant()
: If this unit is dimensionless, return its scalar quantity.

  Direct use of this method is not recommended. It is generally better to use the to or get_converter methods instead.

decompose()
: Return a unit object composed of only irreducible units.

  Parameters
  None :

Returns

unit : CompositeUnit object

New object containing only irreducible unit objects.

PrefixUnit

class astropy.units.core.PrefixUnit(*st*, *represents=None*, *register=False*, *doc=None*, *format=None*)

Bases: astropy.units.core.Unit

A unit that is simply a SI-prefixed version of another unit.

For example, mm is a PrefixUnit of .001 * m.

The constructor is the same as for Unit.

UnrecognizedUnit

class astropy.units.core.UnrecognizedUnit(*st*)

Bases: astropy.units.core.IrreducibleUnit

A unit that did not parse correctly. This allows for roundtripping it as a string, but no unit operations actually work on it.

Parameters

st : str

The name of the unit.

Methods Summary

get_format_name(format)
is_equivalent(other[, equivalencies])
get_converter(other[, equivalencies])
to_string([format])

Methods Documentation

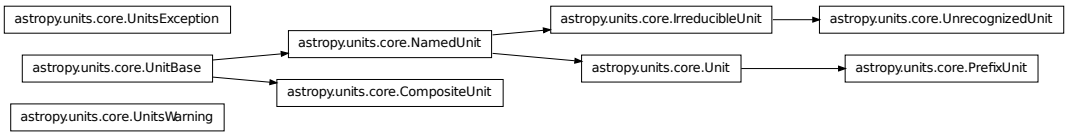
get_format_name(*format*)

is_equivalent(*other*, *equivalencies=[]*)

get_converter(*other*, *equivalencies=[]*)

to_string(*format=u'generic'*)

Class Inheritance Diagram



astropy.units.format Module

A collection of different unit formats.

Functions

get_format([format])	Get a formatter by name.

get_format

astropy.units.format.get_format(*format=None*)

Get a formatter by name.

Parameters

format : str or astropy.units.format.Base instance or subclass

The name of the format, or the format instance or subclass itself.

Returns

format : astropy.units.format.Base instance

The requested formatter.

Classes

Generic()	A "generic" format.
CDS()	Support the Centre de Données astronomiques de Strasbourg Standards for Astronomical Catalogues 2.0 format.
Console()	Output-only format for to display pretty formatting at the console.
Fits()	The FITS standard unit format.
Latex()	Output LaTeX to display the unit based on IAU style guidelines.
Unicode()	Output-only format for to display pretty formatting at the console using Unicode characters.
Unscaled()	A format that doesn't display the scale part of the unit, other than that, it is identical to the Generic format.
VOUnit()	The proposed IVOA standard for units used by the VO.

Generic

class astropy.units.format.generic.Generic

Bases: astropy.units.format.base.Base

A "generic" format.

The syntax of the format is based directly on the FITS standard, but instead of only supporting the units that

FITS knows about, it supports any unit available in the `astropy.units` namespace.

Methods Summary

`parse`(s)	
`to_string`(unit)	

Methods Documentation

`parse`(*s*)

`to_string`(*unit*)

CDS

class `astropy.units.format.cds.CDS`

Bases: `astropy.units.format.base.Base`

Support the Centre de Données astronomiques de Strasbourg Standards for Astronomical Catalogues 2.0 format. This format is used by VOTable up to version 1.2.

Methods Summary

`parse`(s)	
`to_string`(unit)	

Methods Documentation

`parse`(*s*)

`to_string`(*unit*)

Console

class `astropy.units.format.console.Console`

Bases: `astropy.units.format.base.Base`

Output-only format for to display pretty formatting at the console.

For example:

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

Methods Summary

to_string(unit)

Methods Documentation

to_string(*unit*)

Fits

class astropy.units.format.fits.Fits

Bases: astropy.units.format.generic.Generic

The FITS standard unit format.

This supports the format defined in the Units section of the FITS Standard.

Attributes Summary

name	unicode(string [, encoding[, errors]]) -> object

Methods Summary

to_string(unit)

Attributes Documentation

name = **u'fits'**

Methods Documentation

to_string(*unit*)

Latex

class astropy.units.format.latex.Latex

Bases: astropy.units.format.base.Base

Output LaTeX to display the unit based on IAU style guidelines.

Attempts to follow the IAU Style Manual.

Methods Summary

to_string(unit)

Methods Documentation

to_string(*unit*)

Unicode

class astropy.units.format.unicode_format.Unicode

Bases: astropy.units.format.console.Console

Output-only format for to display pretty formatting at the console using Unicode characters.

For example:

```
>>> print u.Ry.decompose().to_string('unicode')
           m² kg
2.18×10⁻¹⁸ ─────
            s²
```

Unscaled

class astropy.units.format.generic.Unscaled

Bases: astropy.units.format.generic.Generic

A format that doesn't display the scale part of the unit, other than that, it is identical to the Generic format.

This is used in some error messages where the scale is irrelevant.

VOUnit

class astropy.units.format.vounit.VOUnit

Bases: astropy.units.format.generic.Generic

The proposed IVOA standard for units used by the VO.

This is an implementation of proposed IVOA standard for units.

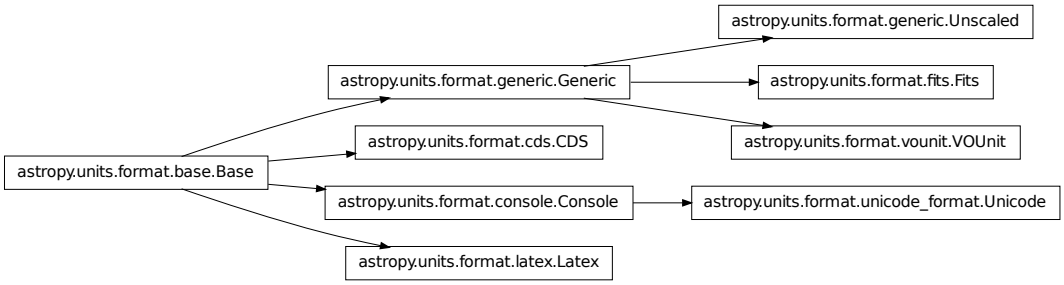
Methods Summary

to_string(unit)	

Methods Documentation

to_string(*unit*)

Class Inheritance Diagram



astropy.units.si Module

This package defines the SI units. They are also available in the `astropy.units` namespace.

Table 1.29: Available Units

Unit	Description	Represents	Aliases
A	ampere: base unit of electric current in SI		ampere, a
a	annum (a)	3.1557×10^{7} s	annum
adu	adu		
Angstrom	ångström: 10 ** -10 m	10^{-1} nm	AA, angst
arcmin	arc minute: angular measurement	1.6667×10^{-2} °	arcminut
arcsec	arc second: angular measurement	2.7778×10^{-4} °	arcsecon
beam	beam		
bin	bin		
bit	bit		
byte	byte		
C	coulomb: electric charge	A s	coulomb
cd	candela: base unit of luminous intensity in SI		candela
chan	chan		
ct	count (ct)		count
d	day (d)	2.4×10^{1} h	day
deg	degree: angular measurement 1/360 of full rotation	1.7453×10^{-2} rad	degree
eV	Electron Volt	1.6022×10^{-19} J	electron
F	Farad: electrical capacitance	$\frac{\mathrm{C}}{\mathrm{V}}$	Farad, fa
fortnight	fortnight	2 wk	
g	gram (g)	10^{-3} kg	gram
H	Henry: inductance	$\frac{\mathrm{Wb}}{\mathrm{A}}$	Henry, he
h	hour (h)	3.6×10^{3} s	hour
Hz	Frequency	$\frac{1}{\mathrm{s}}$	Hertz, he
J	Joule: energy	N m	Joule, jo
K	Kelvin: temperature with a null point at absolute zero.		Kelvin
kg	kilogram: base unit of mass in SI.		kilogram
l	liter: metric unit of volume	10^{3} cm$^{3.0}$	L, liter
lm	lumen: luminous flux	cd sr	lumen

Table 1.29 – continued from previous page

Unit	Description	Represents	Aliases
lx	lux: luminous emittence	$\frac{\mathrm{lm}}{\mathrm{m}^2}$	lux
m	meter: base unit of length in SI		meter
mas	milliarc second: angular measurement	2.7778×10^{-7} °	
micron	micron: alias for micrometer (um)	μm	
min	minute (min)	6×10^{1} s	minute
mol	mole: amount of a chemical substance in SI.		mole
N	Newton: force	$\frac{\mathrm{kg\ m}}{\mathrm{s}^2}$	Newton, n
Ohm	Ohm: electrical resistance	$\frac{\mathrm{V}}{\mathrm{A}}$	ohm
Pa	Pascal: pressure	$\frac{\mathrm{J}}{\mathrm{m}^3}$	Pascal, p
ph	photon (ph)		photon
pix	pixel (pix)		pixel
rad	radian: angular measurement of the ratio between the length on an arc and its radius		radian
s	second: base unit of time in SI.		second
S	Siemens: electrical conductance	$\frac{\mathrm{A}}{\mathrm{V}}$	Siemens,
sday	Sidereal day (sday) is the time of one rotation of the Earth.	8.6164×10^{4} s	
sr	steradian: unit of solid angle in SI	rad^2	steradia
t	Metric tonne	10^{3} kg	tonne
T	Tesla: magnetic flux density	$\frac{\mathrm{Wb}}{\mathrm{m}^2}$	Tesla, te
V	Volt: electric potential or electromotive force	$\frac{\mathrm{J}}{\mathrm{C}}$	Volt, vol
vox	voxel (vox)		voxel
W	Watt: power	$\frac{\mathrm{J}}{\mathrm{s}}$	Watt, wat
Wb	Weber: magnetic flux	V s	Weber, we
wk	week (wk)	7 d	week
yr	year (yr)	3.1557×10^{7} s	year

astropy.units.cgs Module

This package defines the CGS units. They are also available in the top-level `astropy.units` namespace.

Table 1.30: Available Units

Unit	Description	Represents	Aliases	SI Prefixes
Ba	Barye: CGS unit of pressure	$\frac{\mathrm{g}}{\mathrm{cm\ s}^2}$	Barye, barye	N
D	Debye: CGS unit of electric dipole moment	3.3333×10^{-30} m C	Debye, debye	N
dyn	dyne: CGS unit of force	$\frac{\mathrm{g\ cm}}{\mathrm{s}^2}$	dyne	N
erg	erg: CGS unit of energy	$\frac{\mathrm{cm}^2\ \mathrm{g}}{\mathrm{s}^2}$		N
Fr	Franklin (Fr)	$\frac{\mathrm{cm}^{3/2}\ \mathrm{g}^{1/2}}{\mathrm{s}}$	Franklin, statcoulomb, statC, esu	N
G	Gauss: CGS unit for magnetic field	10^{-4} T	Gauss, gauss	N
Gal	Gal: CGS unit of acceleration	$\frac{\mathrm{cm}}{\mathrm{s}^2}$	gal	N
k	kayser: CGS unit of wavenumber	$\frac{1}{\mathrm{cm}}$	Kayser, kayser	N
P	poise: CGS unit of dynamic viscosity	$\frac{\mathrm{g}}{\mathrm{cm\ s}}$	poise	N
St	stokes: CGS unit of kinematic viscosity	$\frac{\mathrm{cm}^2}{\mathrm{s}}$	stokes	N

astropy.units.astrophys Module

This package defines the astrophysics-specific units. They are also available in the `astropy.units` namespace.

The `mag` unit is provided for compatibility with the FITS unit string standard. However, it is not very useful as-is since it is "orphaned" and can not be converted to any other unit. A future astropy magnitudes library is planned to address this shortcoming.

Table 1.31: Available Units

Unit	Description	Represents	Aliases	SI Prefixes
`AU`	astronomical unit: approximately the mean Earth–Sun distance.	1.496×10^{11} m	`au`	N
`barn`	barn: unit of area used in HEP	10^{-28} m^2		N
`Jy`	Jansky: spectral flux density	$10^{-26} \frac{\mathrm{W}}{\mathrm{Hz\ m^2}}$	`Jansky`, `jansky`	N
`lyr`	Light year	9.4607×10^{15} m	`lightyear`	N
`M_e`	Electron mass	9.1094×10^{-31} kg		N
`M_p`	Proton mass	1.6726×10^{-27} kg		N
`mag`	Stellar magnitude.			Y
`pc`	parsec: approximately 3.26 light-years.	3.0857×10^{16} m	`parsec`	N
`R`	Rayleigh: photon flux	$7.9577 \times 10^{8} \frac{\mathrm{ph}}{\mathrm{s\ sr\ m^2}}$	`Rayleigh`, `rayleigh`	N
`Ry`	Rydberg: Energy of a photon whose wavenumber is the Rydberg constant	1.3606×10^{1} eV	`rydberg`	N
`solLum`	Solar luminance	3.846×10^{26} W	`L_sun`	N
`solMass`	Solar mass	1.9891×10^{30} kg	`M_sun`	N
`solRad`	Solar radius	6.9551×10^{8} m	`R_sun`	N
`Sun`	Sun			N
`u`	Unified atomic mass unit	1.6605×10^{-27} kg	`Da`, `Dalton`	N

astropy.units.imperial Module

This package defines colloquially used Imperial units. They are also available in the `astropy.units` namespace.

Table 1.32: Available Units

Unit	Description	Represents	Aliases	SI Prefixes
ac	International acre	4.356×10^{4} ft^2	acre	N
BTU	British thermal unit	1.0551 kJ	btu	N
cal	Thermochemical calorie: pre-SI metric unit of energy	4.184 J	calorie	N
cup	U.S.	5×10^{-1} pint		N
foz	U.S.	1.25×10^{-1} cup	fluid_oz, fluid_ounce	N
ft	International foot	1.2×10^{1} inch	foot	N
gallon	U.S.	3.7854 l		N
hp	Electrical horsepower	7.457×10^{2} W	horsepower	N
inch	International inch	2.54 cm		N
kcal	Calorie: colloquial definition of Calorie	10^{3} cal	Cal, Calorie, kilocal, kilocalorie	N
lb	International avoirdupois pound	1.6×10^{1} oz	pound	N
mi	International mile	5.28×10^{3} ft	mile	N
oz	International avoirdupois ounce	2.835×10^{1} g	ounce	N
pint	U.S.	5×10^{-1} quart		N
quart	U.S.	2.5×10^{-1} gallon		N
tbsp	U.S.	5×10^{-1} foz	tablespoon	N
ton	International avoirdupois ton	2×10^{3} lb		N
tsp	U.S.	3.3333×10^{-1} tbsp	teaspoon	N
yd	International yard	3 ft	yard	N

astropy.units.equivalencies Module

A set of standard astronomical equivalencies.

Functions

spectral()	Returns a list of equivalence pairs that handle spectral wavelength, frequency, and energy equiva
spectral_density(sunit, sfactor)	Returns a list of equivalence pairs that handle spectral density with regard to wavelength and fre

spectral

astropy.units.equivalencies.spectral()

Returns a list of equivalence pairs that handle spectral wavelength, frequency, and energy equivalences.

Allows conversions between wavelength units, frequency units and energy units as they relate to light.

spectral_density

astropy.units.equivalencies.spectral_density(*sunit*, *sfactor*)

Returns a list of equivalence pairs that handle spectral density with regard to wavelength and frequency.

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.

Classes

Quantity(value, unit)	A Quantity represents a number with some associated unit.

Quantity

class astropy.units.quantity.Quantity(*value*, *unit*)

Bases: object

A Quantity represents a number with some associated unit.

Parameters

value : number

The numerical value of this quantity in the units given by unit.

unit : UnitBase instance, str

An object that represents the unit associated with the input value. Must be an UnitBase object or a string parseable by the units package.

Raises

TypeError :

If the value provided is not a Python numeric type.

TypeError :

If the unit provided is not either a Unit object or a parseable string unit.

Attributes Summary

isscalar	True if the value of this quantity is a scalar, or False if it is an array-like object.
unit	A UnitBase object representing the unit of this quantity.
decomposed_unit	Generates a new Quantity with the units decomposed.
cgs	Returns a copy of the current Quantity instance with CGS units.
value	The numerical value of this quantity.
si	Returns a copy of the current Quantity instance with SI units.

Methods Summary

to(unit)	Returns a new Quantity object with the specified units.
copy()	Return a copy of this Quantity instance

Attributes Documentation

isscalar

True if the value of this quantity is a scalar, or False if it is an array-like object.

Note: This is subtly different from numpy.isscalar in that numpy.isscalar returns False for a zero-dimensional array (e.g. np.array(1)), while this is True in that case.

unit

A UnitBase object representing the unit of this quantity.

decomposed_unit
: Generates a new Quantity with the units decomposed. Decomposed units have only irreducible units in them (see astropy.units.UnitBase.decompose).

Returns

newq : Quantity

A new object equal to this quantity with units decomposed.

cgs
: Returns a copy of the current Quantity instance with CGS units. The value of the resulting object will be scaled.

value
: The numerical value of this quantity.

si
: Returns a copy of the current Quantity instance with SI units. The value of the resulting object will be scaled.

Methods Documentation

to(*unit*)
: Returns a new Quantity object with the specified units.

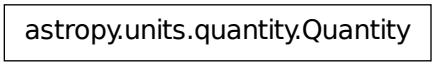
Parameters

unit : UnitBase instance, str

An object that represents the unit to convert to. Must be an UnitBase object or a string parseable by the units package.

copy()
: Return a copy of this Quantity instance

Class Inheritance Diagram



1.5.6 Acknowledgments

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.

1.6 Time and Dates (astropy.time)

1.6.1 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) and time representations (e.g. JD, MJD, ISO 8601) that are used in astronomy. It uses Cython to wrap the C language SOFA time and calendar routines. All time scale conversions are done by Cython vectorized versions of the SOFA routines and are fast and memory efficient.

1.6.2 Getting Started

The basic 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.

```
>>> from astropy.time import Time


>>> times = ['1999-01-01 00:00:00.123456789', '2010-01-01 00:00:00']
>>> t = Time(times, format='iso', scale='utc')
>>> t
<Time object: scale='utc' format='iso' vals=['1999-01-01 00:00:00.123' '2010-01-01 00:00:00.000']>
```

The format argument specifies how to interpret the input values, e.g. ISO or JD or Unix time. The scale argument specifies the time scale for the values, e.g. UTC or TT or UT1.

Now let's 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.        ])
```

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='iso' vals=['1999-01-01 00:01:04.307' '2010-01-01 00:01:06.184']>
>>> t2.jd
array([ 2451179.5007443 ,  2455197.50076602])
```

Note that both the ISO and JD representations of t2 are different than for t because they are expressed relative to the TT time scale.

1.6.3 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.

Once a `Time` object is created it cannot be altered internally. In code lingo it is immutable. In particular the common operation of "converting" to a different time scale is always performed by returning a copy of the original `Time` object which has been converted to the new time scale.

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 easily adapted and extended by users for specialized time formats not supplied in `astropy.time`.

Format	Class
byear	`TimeBesselianEpoch`
byear_str	`TimeBesselianEpochString`
cxcsec	`TimeCxcSec`
iso	`TimeISO`
isot	`TimeISOT`
jd	`TimeJD`
jyear	`TimeJulianEpoch`
jyear_str	`TimeJulianEpochString`
mjd	`TimeMJD`
unix	`TimeUnix`
yday	`TimeYearDayTime`

Subformat The time format classes `TimeISO`, `TimeISO`, and `TimeYearDayTime` support the concept of subformats. This allows for variations on the basic theme of a format in both the input string parsing and the output.

The supported subformats are `date_hms`, `date_hm`, and `date`. The table below illustrates these subformats for `iso` 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
`yday`	date_hms	2001:032:03:04:05.678
`yday`	date_hm	2001:032:03:04
`yday`	date	2001:032

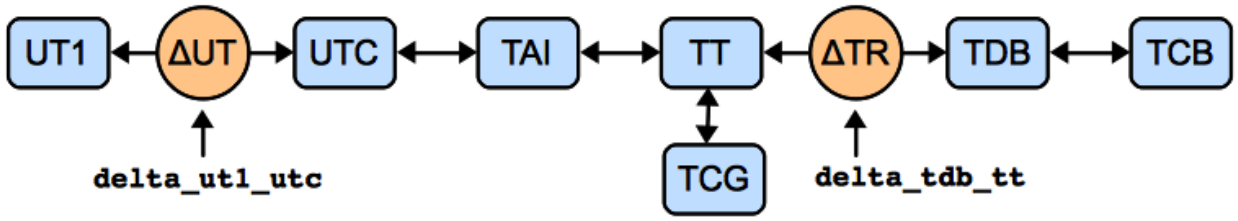
Time Scale

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" [1]. See also [2] and [3].

```
>>> Time.SCALES
('tai', 'tcb', 'tcg', 'tdb', 'tt', 'ut1', 'utc')
```

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)

The system of transformation between supported time scales 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.

```
>>> from astropy.time import Time
>>> t = Time(100.0, format='mjd', scale='utc')
>>> t.jd
2400100.5
>>> t = Time([100.0, 200.0], format='mjd', scale='utc')
>>> t.jd
array([ 2400100.5,  2400200.5])
```

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 a list of strings, in which 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.

[1] Wikipedia time standard article
[2] SOFA Time Scale and Calendar Tools (PDF)
[3] http://www.ucolick.org/~sla/leapsecs/timescales.html

```
>>> t = Time('2010-01-02 01:02:03', scale='utc')
>>> 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 SOFA C-library implementation. The SOFA routines take care throughout to maintain overall precision of the double pair. The user is free to choose the way in which total JD is distributed between the two values.

The internal JD pair is available via the jd1 and jd2 attributes. Notice in the example below that when converting from UTC to TAI, the small offset is placed in the jd2 value thus maintaining the highest numeric precision:

```
>>> t = Time('2010-01-01 00:00:00', scale='utc')
>>> t.jd1, t.jd2
(2455197.5, 0.0)
>>> t2 = t.tai
>>> t2.jd1. t2.jd2
(2455197.5, 0.0003935185185185185)
```

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 string input times

out_subfmt
: [str] Unix glob to select subformats for outputting string times

lat
: [float, optional] Earth latitude of observer

lon
: [float, optional] Earth longitude of observer

val

The `val` argument is the only argument that is always required when creating a `Time` object. This 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.

In most situations one also needs to specify the time scale via the `scale` argument. The `Time` class will never guess the time scale, so a simple example would be:

```
>>> t = Time('2010-01-01 00:00:00', scale='utc')
>>> t2 = Time(50100.0, format='mjd', scale='tt')
```

val2

The `val2` argument is available for specialized situations where extremely 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 the internal time values. The exact interpretation of `val2` is determined by the input format class. As of this release all string-valued formats ignore `val2` and all numeric inputs effectively add the two values in a way that maintains the highest precision. Example:

```
>>> t = Time(100.0, 0.000001, format='mjd', scale='tt')
>>> t.jd, t.jd1, t.jd2
(2400100.500001, 2400100.5, 1e-06)
```

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 'cxcsec' (`TimeCxcSec`) and 'unix' (`TimeUnix`). These formats represent the duration in SI seconds since a fixed instant in time which is independent of time scale.

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 SOFA handles fractional seconds. In practice this should should not be an issue.

```
>>> t = Time('B1950.0', scale='utc', 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 string 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 heterogeous subformat:

```
>>> Time(['2000:001', '2000:002:03:04', '2001:003:04:05:06.789'], scale='utc')
<Time object: scale='utc' format='yday'
 vals=['2000:001:00:00:00.000' '2000:002:03:04:00.000' '2001:003:04:05:06.789']>
```

One can explicitly specify in_subfmt in order to strictly require a certain subformat:

```
>>> t = Time('2000:002:03:04', scale='utc', in_subfmt='date_hm')
>>> t = Time('2000:002', scale='utc', in_subfmt='date_hm')
ERROR: ValueError: Input values did not match any of format classes
['iso', 'isot', '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', scale='utc', out_subfmt='date').iso
'2000-01-01'
>>> Time('2000-01-01 02:03:04', scale='utc', out_subfmt='date_hms').iso
'2000-01-01 02:03:04.000'
>>> Time('2000-01-01 02:03:04', scale='utc', out_subfmt='date*').iso
'2000-01-01 02:03:04.000'
```

lat and lon

These optional parameters specify the observer latitude and longitude in decimal degrees. They default to 0.0 and are used for time scales that are sensitive to observer position. Currently the only time scale for which this applies is TDB, which relies on the SOFA routine iauDtdb to determine the time offset between TDB and TT.

Using Time objects

There are three basic operations available with Time objects:

- 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.
- Do time arithmetic involving Time and/or TimeDelta objects.

Get representation

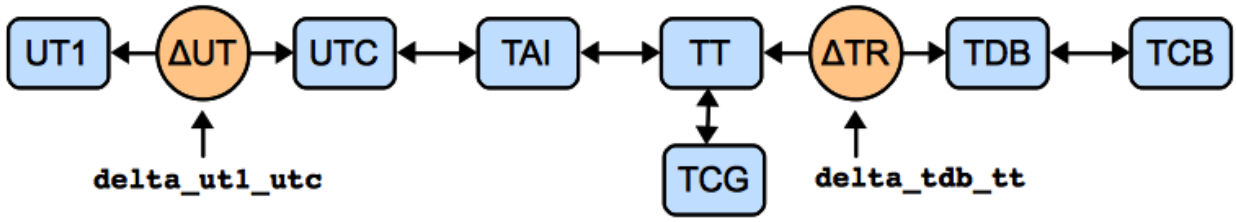
Instants of time can be represented in different ways, for instance as an ISO-format date string (`'1999-07-23 04:31:00'`) or seconds since 1998.0 (`49091460.0`) or Modified Julian Date (`51382.187451574`).

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.cxcsec    # SI seconds since 1998.0 (TT)
378691266.184
```

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:

```
>>> t = Time('2010-01-01 00:00:00', format='iso', scale='utc')
>>> t.tt        # TT scale
<Time object: scale='tt' format='iso' vals=2010-01-01 00:01:06.184>
>>> t.tai
<Time object: scale='tai' format='iso' vals=2010-01-01 00:00:34.000>
```

In this process the `format` and other object attributes like `lat`, `lon`, and `precision` are also propagated to the new object.

As noted in the `Time object basics` section, a `Time` object is immutable and the internal time values cannot be altered once the object is created. 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.

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'
```

In the case of the TDB to TT offset, most users need only provide the `lat` and `lon` values when creating the `Time` object. If the `delta_tdb_tt` attribute is not explicitly set then the SOFA C-library routine `iauDtdb` will be used to compute the TDB to TT offset. Note that `lat` and `lon` are initialized to 0.0 by default, so those defaults will be used if they are not provided.

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 auxilliary information (latitude and longitude).

```
>>> lat = 19.48125
>>> lon = -155.933222
>>> t = Time('2006-01-15 21:24:37.5', format='iso', scale='utc',
...          lat=lat, lon=lon, precision=6)
>>> t.delta_ut1_utc = 0.3341  # Explicitly set one part of the transformation
>>> t.utc.iso
'2006-01-15 21:24:37.500000'
>>> t.ut1.iso
'2006-01-15 21:24:37.834100'
>>> t.tai.iso
'2006-01-15 21:25:10.500000'
>>> t.tt.iso
'2006-01-15 21:25:42.684000'
>>> t.tcg.iso
'2006-01-15 21:25:43.322690'
>>> t.tdb.iso
'2006-01-15 21:25:42.683799'
>>> t.tcb.iso
'2006-01-15 21:25:56.893378'
```

Time Deltas

Simple time arithmetic is supported using via the `TimeDelta` class. The following operations are available:

- Create a TimeDelta explicitly by instantiating a class object
- Create a TimeDelta by subtracting two Times
- 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

The `TimeDelta` class is derived from the `Time` class and shares many of its properties. The key difference is that the time scale is always TAI so that all time deltas are referenced to a uniform Julian Day which is exactly 86400 standard SI seconds.

The available time formats are:

Format	Class
sec	TimeDeltaSec
jd	TimeDeltaJD

Examples

Use of the TimeDelta object is easily illustrated in the few examples below:

```
>>> t1 = Time('2010-01-01 00:00:00', scale='utc')
>>> t2 = Time('2010-02-01 00:00:00', scale='utc')
>>> dt = t2 - t1  # Difference between two Times
>>> dt
<TimeDelta object: scale='tai' format='jd' vals=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' vals=2010-01-31 23:59:10.000>

>>> dt + dt2
<TimeDelta object: scale='tai' format='jd' vals=31.0005787037>
```

1.6.4 Reference/API

astropy.time.core Module

The astropy.time package provides functionality for manipulating times and dates. Specific emphasis is placed on supporting time scales (e.g. UTC, TAI, UT1) and time representations (e.g. JD, MJD, ISO 8601) that are used in astronomy.

Classes

Time(val[, val2, format, scale, precision, ...])	Represent and manipulate times and dates for astronomy.
TimeDelta(val[, val2, format, scale, copy])	Represent the time difference between two times.
TimeFormat(val1, val2, scale, precision, ...)	Base class for time representations.
TimeJD(val1, val2, scale, precision, ...[, ...])	Julian Date time format
TimeMJD(val1, val2, scale, precision, ...[, ...])	Modified Julian Date time format
TimeFromEpoch(val1, val2, scale, precision, ...)	Base class for times that represent the interval from a particular epoch as a floati
TimeUnix(val1, val2, scale, precision, ...)	Unix time: seconds from 1970-01-01 00:00:00 UTC.
TimeCxcSec(val1, val2, scale, precision, ...)	Chandra X-ray Center seconds from 1998-01-01 00:00:00 TT
TimeString(val1, val2, scale, precision, ...)	Base class for string-like time represetations.
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...".
TimeYearDayTime(val1, val2, scale, ...[, ...])	Year, day-of-year and time as "YYYY:DOY:HH:MM:SS.sss...".
TimeEpochDate(val1, val2, scale, precision, ...)	Base class for support floating point Besselian and Julian epoch dates
TimeBesselianEpoch(val1, val2, scale, ...[, ...])	Besselian Epoch year as floating point value(s) like 1950.0
TimeJulianEpoch(val1, val2, scale, ...[, ...])	Julian Epoch year as floating point value(s) like 2000.0
TimeDeltaFormat(val1, val2, scale, ...[, ...])	Base class for time delta representations

Table 1.37 – continued from previous page

TimeDeltaSec(val1, val2, scale, precision, ...)	Time delta in SI seconds
TimeDeltaJD(val1, val2, scale, precision, ...)	Time delta in Julian days (86400 SI seconds)
ScaleValueError	
OperandTypeError(left, right)	
TimeEpochDateString(val1, val2, scale, ...)	Base class to support string Besselian and Julian epoch dates such as 'B1950.0'
TimeBesselianEpochString(val1, val2, scale, ...)	Besselian Epoch year as string value(s) like 'B1950.0'
TimeJulianEpochString(val1, val2, scale, ...)	Julian Epoch year as string value(s) like 'J2000.0'

Time

class astropy.time.core.Time(*val*, *val2=None*, *format=None*, *scale=None*, *precision=None*, *in_subfmt=None*, *out_subfmt=None*, *lat=0.0*, *lon=0.0*, *copy=False*)

Bases: object

Represent and manipulate times and dates for astronomy.

A Time object is initialized with one or more times in the `val` argument. The input times in `val` must conform to the specified `format` and must correspond to the specified time `scale`. The optional `val2` time input should be supplied only for numeric input formats (e.g. JD) where very high precision (better than 64-bit precision) is required.

Parameters

val : sequence, str, number, or Time object

Value(s) to initialize the time or times.

val2 : sequence, str, or number; optional

Value(s) to initialize the time or times

format : str, optional

Format of input value(s)

scale : str, optional

Time scale of input value(s)

opt : dict, optional

options

lat : float, optional

Earth latitude of observer (decimal degrees)

lon : float, optional

Earth longitude of observer (decimal degrees)

copy : bool, optional

Make a copy of the input values

Attributes Summary

delta_tdb_tt	
FORMATS	
scale	Time scale

Continued on next page

Table 1.38 – continued from previous page

val	Time value(s) in current format
SCALES	tuple() -> empty tuple
out_subfmt	Unix wildcard pattern to select subformats for outputting times
jd1	First of the two doubles that internally store time value(s) in JD
jd2	Second of the two doubles that internally store time value(s) in JD
delta_ut1_utc	Get SOFA DUT arg = UT1 - UTC.
format	Time format
precision	Decimal precision when outputting seconds as floating point (int value between 0 and 9 inclusive).
vals	Time values in current format as a numpy array
in_subfmt	Unix wildcard pattern to select subformats for parsing string input

Methods Summary

replicate([format, copy])	Return a replica of the Time object, optionally changing the format.
copy([format])	Return a fully independent copy the Time object, optionally changing the format.

Attributes Documentation

delta_tdb_tt
: TDB - TT time scale offset

FORMATS = **{'mjd': <class 'astropy.time.core.TimeMJD'>, 'cxcsec': <class 'astropy.time.core.TimeCxcSec'>, 'jyear': <class**
: Dict of time formats

scale
: Time scale

val
: Time value(s) in current format

SCALES = **('tai', 'tcb', 'tcg', 'tdb', 'tt', 'ut1', 'utc')**
: List of time scales

out_subfmt
: Unix wildcard pattern to select subformats for outputting times

jd1
: First of the two doubles that internally store time value(s) in JD

jd2
: Second of the two doubles that internally store time value(s) in JD

delta_ut1_utc
: UT1 - UTC time scale offset

format
: Time format

precision
: Decimal precision when outputting seconds as floating point (int value between 0 and 9 inclusive).

vals
: Time values in current format as a numpy array

in_subfmt
: Unix wildcard pattern to select subformats for parsing string input times

Methods Documentation

replicate(*format=None*, *copy=False*)

Return a replica of the Time object, optionally changing the format.

If `format` is supplied then the time format of the returned Time object will be set accordingly, otherwise it will be unchanged from the original.

If `copy` is set to True then a full copy of the internal time arrays will be made. By default the replica will use a reference to the original arrays when possible to save memory. The internal time arrays are normally not changeable by the user so in most cases it should not be necessary to set `copy` to True.

The convenience method copy() is available in which `copy` is True by default.

Parameters

format : str, optional

Time format of the replica.

copy : bool, optional

Return a true copy instead of using references where possible.

Returns

tm: Time object :

Replica of this object

copy(*format=None*)

Return a fully independent copy the Time object, optionally changing the format.

If `format` is supplied then the time format of the returned Time object will be set accordingly, otherwise it will be unchanged from the original.

In this method a full copy of the internal time arrays will be made. The internal time arrays are normally not changeable by the user so in most cases the `replicate()` method should be used.

Parameters

format : str, optional

Time format of the copy.

Returns

tm: Time object :

Copy of this object

TimeDelta

class astropy.time.core.TimeDelta(*val*, *val2=None*, *format=None*, *scale=None*, *copy=False*)

Bases: astropy.time.core.Time

Represent the time difference between two times.

A TimeDelta object is initialized with one or more times in the `val` argument. The input times in `val` must conform to the specified `format`. The optional `val2` time input should be supplied only for numeric input formats (e.g. JD) where very high precision (better than 64-bit precision) is required.

Parameters

val : numpy ndarray, list, str, number, or `TimeDelta` object

Data to initialize table.

val2 : numpy ndarray, list, str, or number; optional

Data to initialize table.

format : str, optional

Format of input value(s)

copy : bool, optional

Make a copy of the input values

Attributes Summary

SCALES	tuple() -> empty tuple
FORMATS	

Attributes Documentation

SCALES = **('tai',)**
List of time delta scales

FORMATS = **{'jd': <class 'astropy.time.core.TimeDeltaJD'>, 'sec': <class 'astropy.time.core.TimeDeltaSec'>}**
Dict of time delta formats

TimeFormat

class astropy.time.core.TimeFormat(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: object

Base class for time representations.

Parameters

val1 : numpy ndarray, list, str, or number

Data to initialize table.

val2 : numpy ndarray, list, str, or number; optional

Data to initialize table.

scale : str

Time scale of input value(s)

precision : int

Precision for seconds as floating point

in_subfmt : str

Select subformat for inputting string times

out_subfmt : str

Select subformat for outputting string times

from_jd : bool

If true then val1, val2 are jd1, jd2

Attributes Summary

scale	Time scale
vals	Return time representation from internal jd1 and jd2.

Methods Summary

set_jds(val1, val2)	Set internal jd1 and jd2 from val1 and val2.

Attributes Documentation

scale
: Time scale

vals
: Return time representation from internal jd1 and jd2. Must be provided by by derived classes.

Methods Documentation

set_jds(*val1*, *val2*)
: Set internal jd1 and jd2 from val1 and val2. Must be provided by derived classes.

TimeJD

class astropy.time.core.TimeJD(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: astropy.time.core.TimeFormat

Julian Date time format

Attributes Summary

vals	
name	str(object) -> string

Methods Summary

set_jds(val1, val2)

Attributes Documentation

vals

name = **'jd'**

Methods Documentation

set_jds(*val1*, *val2*)

TimeMJD

class astropy.time.core.TimeMJD(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: astropy.time.core.TimeFormat

Modified Julian Date time format

Attributes Summary

vals	
name	str(object) -> string

Methods Summary

set_jds(val1, val2)	

Attributes Documentation

vals

name = **'mjd'**

Methods Documentation

set_jds(*val1*, *val2*)

TimeFromEpoch

class astropy.time.core.TimeFromEpoch(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: astropy.time.core.TimeFormat

Base class for times that represent the interval from a particular epoch as a floating point multiple of a unit time interval (e.g. seconds or days).

Attributes Summary

vals	

Methods Summary

set_jds(val1, val2)

Attributes Documentation

vals

Methods Documentation

set_jds(*val1*, *val2*)

TimeUnix

class astropy.time.core.TimeUnix(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: astropy.time.core.TimeFromEpoch

Unix time: seconds from 1970-01-01 00:00:00 UTC.

NOTE: this quantity is not exactly unix time and differs from the strict POSIX definition by up to 1 second on days with a leap second. POSIX unix time actually jumps backward by 1 second at midnight on leap second days while this class value is monotonically increasing at 86400 seconds per UTC day.

Attributes Summary

name	str(object) -> string
epoch_val2	
epoch_format	str(object) -> string
epoch_val	str(object) -> string
unit	float(x) -> floating point number
epoch_scale	str(object) -> string

Attributes Documentation

name = **'unix'**

epoch_val2 = **None**

epoch_format = **'iso'**

epoch_val = **'1970-01-01 00:00:00'**

unit = **1.1574074074074073e-05**

epoch_scale = **'utc'**

TimeCxcSec

class astropy.time.core.TimeCxcSec(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: astropy.time.core.TimeFromEpoch

Chandra X-ray Center seconds from 1998-01-01 00:00:00 TT

Attributes Summary

name	str(object) -> string
epoch_val2	
epoch_format	str(object) -> string
epoch_val	str(object) -> string
unit	float(x) -> floating point number
epoch_scale	str(object) -> string

Attributes Documentation

name = **'cxcsec'**

epoch_val2 = **None**

epoch_format = **'iso'**

epoch_val = **'1998-01-01 00:00:00'**

unit = **1.1574074074074073e-05**

epoch_scale = **'tt'**

TimeString

class astropy.time.core.TimeString(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: astropy.time.core.TimeFormat

Base class for string-like time represetations.

This class assumes that anything following the last decimal point to the right is a fraction of a second.

This is a reference implementation can be made much faster with effort.

Attributes Summary

vals

Methods Summary

set_jds(val1, val2)	Parse the time strings contained in val1 and set jd1, jd2
str_kwargs()	Generator that yields a dict of values corresponding to the calendar date and time for the internal JD values.

Attributes Documentation

vals

Methods Documentation

set_jds(*val1*, *val2*)
Parse the time strings contained in val1 and set jd1, jd2

str_kwargs()
Generator that yields a dict of values corresponding to the calendar date and time for the internal JD values.

TimeISO

class astropy.time.core.TimeISO(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)
Bases: astropy.time.core.TimeString

ISO 8601 compliant date-time format "YYYY-MM-DD HH:MM:SS.sss...".

The allowed subformats are:

- 'date_hms': date + hours, mins, secs (and optional fractional secs)
- 'date_hm': date + hours, mins
- 'date': date

Attributes Summary

subfmts	tuple() -> empty tuple
name	str(object) -> string

Attributes Documentation

subfmts = **(('date_hms', '%Y-%m-%d %H:%M:%S', '{year:d}-{mon:02d}-{day:02d} {hour:02d}:{min:02d}:{sec:02d}'),**

name = **'iso'**

TimeISOT

class astropy.time.core.TimeISOT(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)
Bases: astropy.time.core.TimeString

ISO 8601 compliant date-time format "YYYY-MM-DDTHH:MM:SS.sss...". This is the same as TimeISO except for a "T" instead of space between the date and time.

The allowed subformats are:

- 'date_hms': date + hours, mins, secs (and optional fractional secs)
- 'date_hm': date + hours, mins
- 'date': date

Attributes Summary

subfmts	tuple() -> empty tuple
name	str(object) -> string

Attributes Documentation

subfmts = **(('date_hms', '%Y-%m-%dT%H:%M:%S', '{year:d}-{mon:02d}-{day:02d}T{hour:02d}:{min:02d}:{sec:02d}**

name = **'isot'**

TimeYearDayTime

class astropy.time.core.TimeYearDayTime(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: astropy.time.core.TimeString

Year, day-of-year and time as "YYYY:DOY:HH:MM:SS.sss...". The day-of-year (DOY) goes from 001 to 365 (366 in leap years).

The allowed subformats are:

- 'date_hms': date + hours, mins, secs (and optional fractional secs)
- 'date_hm': date + hours, mins
- 'date': date

Attributes Summary

subfmts	tuple() -> empty tuple
name	str(object) -> string

Attributes Documentation

subfmts = **(('date_hms', '%Y:%j:%H:%M:%S', '{year:d}:{yday:03d}:{hour:02d}:{min:02d}:{sec:02d}'), ('date_hm', '%**

name = **'yday'**

TimeEpochDate

class astropy.time.core.TimeEpochDate(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: astropy.time.core.TimeFormat

Base class for support floating point Besselian and Julian epoch dates

Attributes Summary

vals	

Methods Summary

set_jds(val1, val2)	

Attributes Documentation

vals

Methods Documentation

set_jds(*val1*, *val2*)

TimeBesselianEpoch

class astropy.time.core.TimeBesselianEpoch(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: astropy.time.core.TimeEpochDate

Besselian Epoch year as floating point value(s) like 1950.0

Attributes Summary

jd_to_epoch	str(object) -> string
name	str(object) -> string
epoch_to_jd	str(object) -> string

Attributes Documentation

jd_to_epoch = **'jd_besselian_epoch'**

name = **'byear'**

epoch_to_jd = **'besselian_epoch_jd'**

TimeJulianEpoch

class astropy.time.core.TimeJulianEpoch(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: astropy.time.core.TimeEpochDate

Julian Epoch year as floating point value(s) like 2000.0

Attributes Summary

jd_to_epoch	str(object) -> string
name	str(object) -> string
epoch_to_jd	str(object) -> string

Attributes Documentation

jd_to_epoch = **'jd_julian_epoch'**

name = **'jyear'**

epoch_to_jd = **'julian_epoch_jd'**

TimeDeltaFormat

class astropy.time.core.TimeDeltaFormat(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: astropy.time.core.TimeFormat

Base class for time delta representations

TimeDeltaSec

class astropy.time.core.TimeDeltaSec(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: astropy.time.core.TimeDeltaFormat

Time delta in SI seconds

Attributes Summary

vals	
name	str(object) -> string

Methods Summary

set_jds(val1, val2)	

Attributes Documentation

vals

name = **'sec'**

Methods Documentation

set_jds(*val1*, *val2*)

TimeDeltaJD
class astropy.time.core.TimeDeltaJD(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: astropy.time.core.TimeDeltaFormat

Time delta in Julian days (86400 SI seconds)

Attributes Summary

vals	
name	str(object) -> string

Methods Summary

set_jds(val1, val2)

Attributes Documentation

vals

name = **'jd'**

Methods Documentation

set_jds(*val1*, *val2*)

ScaleValueError
exception astropy.time.core.ScaleValueError

OperandTypeError
exception astropy.time.core.OperandTypeError(*left*, *right*)

TimeEpochDateString

class astropy.time.core.TimeEpochDateString(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: astropy.time.core.TimeString

Base class to support string Besselian and Julian epoch dates such as 'B1950.0' or 'J2000.0' respectively.

Attributes Summary

vals	

Methods Summary

set_jds(val1, val2)	

Attributes Documentation

vals

Methods Documentation

set_jds(*val1*, *val2*)

TimeBesselianEpochString

class astropy.time.core.TimeBesselianEpochString(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: astropy.time.core.TimeEpochDateString

Besselian Epoch year as string value(s) like 'B1950.0'

Attributes Summary

name	str(object) -> string
epoch_prefix	str(object) -> string
jd_to_epoch	str(object) -> string
epoch_to_jd	str(object) -> string

Attributes Documentation

name = **'byear_str'**

epoch_prefix = **'B'**

jd_to_epoch = **'jd_besselian_epoch'**

epoch_to_jd = **'besselian_epoch_jd'**

TimeJulianEpochString

class astropy.time.core.TimeJulianEpochString(*val1*, *val2*, *scale*, *precision*, *in_subfmt*, *out_subfmt*, *from_jd=False*)

Bases: astropy.time.core.TimeEpochDateString

Julian Epoch year as string value(s) like 'J2000.0'

Attributes Summary

name	str(object) -> string
epoch_prefix	str(object) -> string
jd_to_epoch	str(object) -> string
epoch_to_jd	str(object) -> string

Attributes Documentation

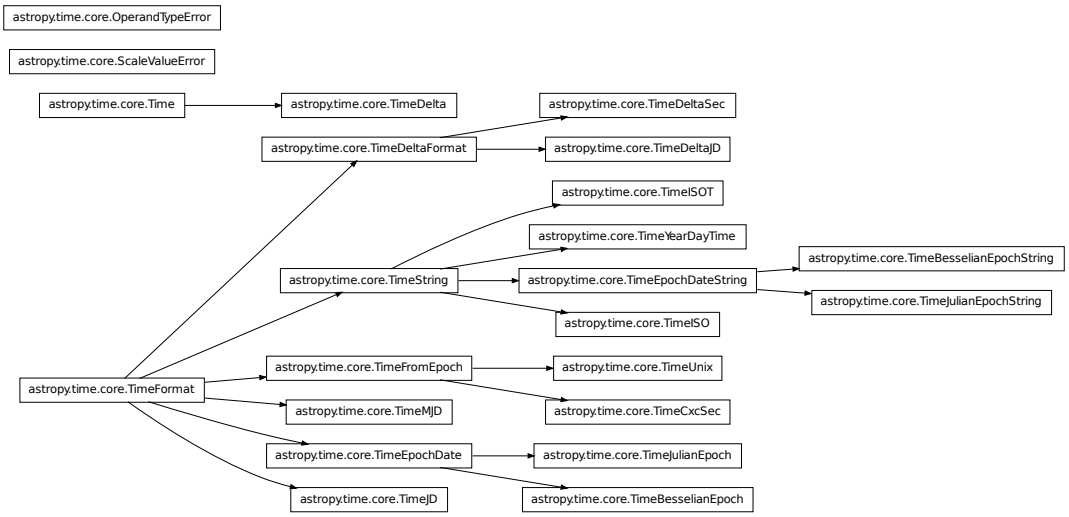
name = **'jyear_str'**

epoch_prefix = **'J'**

jd_to_epoch = **'jd_julian_epoch'**

epoch_to_jd = **'julian_epoch_jd'**

Class Inheritance Diagram



1.6.5 Acknowledgments and Licenses

This package makes use of the SOFA Software ANSI C library. The copyright of the SOFA Software belongs to the Standards Of Fundamental Astronomy Board of the International Astronomical Union. This library is made available under the terms of the SOFA license.

1.7 Astronomical Coordinate Systems (`astropy.coordinates`)

1.7.1 Introduction

The `coordinates` package provides classes for representing celestial coordinates, as well as tools for converting between standard systems in a uniform way.

Note: The current `coordinates` framework only accepts scalar coordinates, i.e. one coordinate per object. In the next release it will be expanded to accept arrays of coordinates.

Warning: `coordinates` is currently a work-in-progress, and thus it is possible there will be significant API changes in later versions of Astropy.

1.7.2 Getting Started

Coordinate objects are instantiated with a flexible and natural approach that supports both numeric angle values and (limited) string parsing:

```
>>> from astropy import coordinates as coord
>>> from astropy import units as u
>>> coord.ICRSCoordinates(ra=10.68458, dec=41.26917, unit=(u.degree, u.degree))
<ICRSCoordinates RA=10.68458 deg, Dec=41.26917 deg>
>>> coord.ICRSCoordinates('00h42m44.3s +41d16m9s')
<ICRSCoordinates RA=10.68458 deg, Dec=41.26917 deg>
```

The individual components of a coordinate are `Angle` objects, and their values are accessed using special attributes:

```
>>> c = coord.ICRSCoordinates(ra=10.68458, dec=41.26917, unit=(u.degree, u.degree))
>>> c.ra
<RA 10.68458 deg>
>>> c.ra.hours
0.7123053333333333
>>> c.ra.hms
(0.0, 42, 44.2992000000001)
>>> c.dec
<Dec 41.26917 deg>
>>> c.dec.radians
0.7202828960652683
```

To convert to some other coordinate system, the easiest method is to use attribute-style access with short names for the built-in systems, but explicit transformations via the `transform_to` method are also available:

```
>>> c.galactic
<GalacticCoordinates l=121.17422 deg, b=-21.57283 deg>
>>> c.transform_to(coord.GalacticCoordinates)
<GalacticCoordinates l=121.17422 deg, b=-21.57283 deg>
```

Distances from the origin (which is system-dependent, but often the Earth center) can also be assigned to a coordinate. This specifies a unique point in 3D space, which also allows conversion to cartesian coordinates:

```
>>> c = coord.ICRSCoordinates(ra=10.68458, dec=41.26917, unit=(u.degree, u.degree), distance=coord.Distance(770, u.kpc))
>>> c.x
568.7128654235232
>>> c.y
107.3008974042025
>>> c.z
507.88994291875713
```

1.7.3 Using `astropy.coordinates`

More details of using `astropy.coordinates` are provided in the following sections:

Working with Angles

The angular components of a coordinate are represented by objects of the `Angle` class. These objects can be instantiated on their own anywhere a representation of an angle is needed, and support a variety of ways of representing the value of the angle:

```
>>> from astropy.coordinates import Angle
>>> a = Angle(1, u.radian)
>>> a
<astropy.coordinates.angles.Angle 57.29578 deg>
>>> a.radians
1
>>> a.degrees
57.29577951308232
>>> a.hours
3.819718634205488
>>> a.hms
(3.0, 49, 10.987083139757061)
>>> a.dms
(57.0, 17, 44.80624709636231)
>>> a.format()
'57d17m44.80625s'
>>> a.format(sep=':')
'57:17:44.80625'
>>> a.format(sep=('deg','m','s'))
'57deg17m44.80625s'
>>> a.format(u.hour)
'3h49m10.98708s'
>>> a.format(u.radian)
'1.0radian'
>>> a.format(u.radian, decimal=True)
'1.0'
```

`Angle` objects can also have bounds. These specify either a limited range in which the angle is valid (if it's <360 degrees), or the limit at which an angle is wrapped back around to 0:

```
>>> Angle(90, unit=u.degree, bounds=(0,180))
<Angle 90.00000 deg>
>>> Angle(-270, unit=u.degree, bounds=(0,180))
<Angle 90.00000 deg>
>>> Angle(181, unit=u.degree, bounds=(0,180))
BoundsError: The angle given falls outside of the specified bounds.
>>> Angle(361, unit=u.degree, bounds=(0,360))
<Angle 1.00000 deg>
```

Angles will also behave correctly for appropriate arithmetic operations:

```
>>> a = Angle(1, u.radian)
>>> a + a
<Angle 114.59156 deg>
>>> a - a
<Angle 0.00000 deg>
>>> a == a
True
>>> a == (a + a)
False
```

Angle objects can also be used for creating coordinate objects:

```
>>> ICRSCoordinates(Angle(1, u.radian), Angle(2, u.radian))
<ICRSCoordinates RA=57.29578 deg, Dec=114.59156 deg>
```

```
>>> ICRSCoordinates(RA(1, u.radian), Dec(2, u.radian))
<ICRSCoordinates RA=57.29578 deg, Dec=114.59156 deg>
```

Creating Coordinate Objects

Creating new coordinate objects is of course crucial to using `coordinates`. The typical way to create a new coordinate object is to directly initialize your preferred coordinate system using standard python class creation, using the name of the class representing that system and a number for the two angles. For example:

```
>>> from astropy.coordinates import ICRSCoordinates, FK4Coordinates, GalacticCoordinates
>>> ICRSCoordinates(187.70592, 12.39112, unit=(u.degree, u.degree))
<ICRSCoordinates RA=187.70592 deg, Dec=12.39112 deg>
>>> FK4Coordinates(187.07317, 12.66715, unit=(u.degree, u.degree))
<FK4Coordinates RA=187.07317 deg, Dec=12.66715 deg>
>>> GalacticCoordinates(-76.22237, 74.49108, unit=(u.degree, u.degree))
<GalacticCoordinates l=-76.22237 deg, b=74.49108 deg>
```

Note that if you do not provide units explicitly, this will fail:

```
>>> ICRSCoordinates(23, 1)
UnitsError: No unit was specified in Angle initializer; the unit parameter should be an object from the  astropy.units mo
```

While the above example uses python numerical types, you can also provide strings to create coordinates. If the `unit` parameter is `(None, None)` (the default), strings will be interpreted using the `Angle` class' parsing scheme, and has a guiding principal of being able to interpret any *unambiguous* string specifying an angle. For the exact rules for how each string is parsed, see the `Angle` documentation. Some examples:

```
>>> ICRSCoordinates("3h36m29.7888s -41d08m15.162342s", unit=(None, None))
<ICRSCoordinates RA=54.12412 deg, Dec=-41.13755 deg>
>>> ICRSCoordinates("3h36m29.7888s -41d08m15.162342s")
<ICRSCoordinates RA=54.12412 deg, Dec=-41.13755 deg>
>>> ICRSCoordinates("14.12412 hours", "-41:08:15.162342 degrees")
<ICRSCoordinates RA=211.86180 deg, Dec=-41.13755 deg>
>>> ICRSCoordinates("14.12412 -41:08:15.162342")
UnitsError: Could not infer Angle units from provided string 14.12412
```

You can also directly specify the units for both to resolve ambiguities in parsing the angle strings:

```
>>> ICRSCoordinates("14.12412 -41:08:15.162342", unit=(u.hour, u.degree))
<ICRSCoordinates RA=211.86180 deg, Dec=-41.13755 deg>
>>> ICRSCoordinates("54:7:26.832 -41:08:15.162342", unit=(u.degree, u.degree))
<ICRSCoordinates RA=54.12412 deg, Dec=-41.13755 deg>
>>> ICRSCoordinates('3 4 5 +6 7 8', unit=(u.hour, u.degree))
<ICRSCoordinates RA=46.02083 deg, Dec=6.11889 deg>
>>> ICRSCoordinates('3h4m5s +6d7m8s', unit=(u.hour, u.degree))
<ICRSCoordinates RA=46.02083 deg, Dec=6.11889 deg>
```

This will also give you an error if you give a string with units that conflict with your desired units:

```
>>> ICRSCoordinates('3d4m5s +6h7m8s', unit=(u.hour, u.degree))
ValueError: parse_hours: Invalid input string, can't parse to HMS. (3d4m5s)
```

One final way to create coordinates is to copy them from an already existing coordinate:

```
>>> i1 = ICRSCoordinates(187.70592, 12.39112, unit=(u.degree, u.degree))
>>> i2 = ICRSCoordinates(i1)
>>> i1
<ICRSCoordinates RA=187.70592 deg, Dec=12.39112 deg>
>>> i2
<ICRSCoordinates RA=187.70592 deg, Dec=12.39112 deg>
```

Separations

The on-sky separation is easily computed with the `separation` method, which computes the great-circle distance (*not* the small-angle approximation):

```
>>> c1 = ICRSCoordinates('5h23m34.5s -69d45m22s')
>>> c2 = ICRSCoordinates('0h52m44.8s -72d49m43s')
>>> sep = c1.separation(c2)
>>> sep
<AngularSeparation 20.74612 deg>
```

The `AngularSeparation` object is a subclass of `Angle`, so it can be accessed in the same ways, along with a few additions:

```
>>> sep.radians
0.36208807374669766
>>> sep.hours
1.383074562513832
>>> sep.arcmins
1244.7671062624488
>>> sep.arcsecs
74686.02637574692
```

Distances and Cartesian Representations

Coordinates can also have line-of-sight distances. If these are provided, a coordinate object becomes a full-fledged point in three-dimensional space. If not (i.e., the `distance` attribute of the coordinate object is `None`), the point is interpreted as lying on the (dimensionless) unit sphere.

The `Distance` class is provided to represent a line-of-sight distance for a coordinate. It must include a length unit to be valid.:

```
>>> from astropy.coordinates import Distance
>>> d = Distance(770)
UnitsError: A unit must be provided for distance.
>>> d = Distance(770, u.kpc)
>>> c = ICRSCoordinates('00h42m44.3s +41d16m9s', distance=d)
>>> c
<ICRSCoordinates RA=10.68458 deg, Dec=41.26917 deg, Distance=7.7e+02 kpc>
```

If a distance is available, the coordinate can be converted into cartesian coordinates using the x/y/z attributes:

```
>>> c.x
568.7128882165681
>>> c.y
107.3009359688103
>>> c.z
507.8899092486349
```

Note: The location of the origin is different for different coordinate systems, but for common celestial coordinate systems it is often the Earth center (or for precision work, the Earth/Moon barycenter).

The cartesian coordinates can also be accessed via the `CartesianCoordinates` object, which has additional capabilities like arithmetic operations:

```
>>> cp = c.cartesian
>>> cp
<CartesianPoints (568.712888217, 107.300935969, 507.889909249) kpc>
>>> cp.x
568.7128882165681
>>> cp.y
107.3009359688103
>>> cp.z
507.8899092486349
>>> cp.unit
Unit("kpc")
>>> cp + cp
<CartesianPoints (1137.42577643, 214.601871938, 1015.7798185) kpc>
>>> cp - cp
<CartesianPoints (0.0, 0.0, 0.0) kpc>
```

This cartesian representation can also be used to create a new coordinate object, either directly or through a `CartesianPoints` object:

```
>>> ICRSCoordinates(x=568.7129, y=107.3009, z=507.8899, unit=u.kpc)
<ICRSCoordinates RA=10.68458 deg, Dec=41.26917 deg, Distance=7.7e+02 kpc>
>>> cp = CartesianPoints(x=568.7129, y=107.3009, z=507.8899, unit=u.kpc)
>>> ICRSCoordinates(cp)
<ICRSCoordinates RA=10.68458 deg, Dec=41.26917 deg, Distance=7.7e+02 kpc>
```

Finally, two coordinates with distances can be used to derive a real-space distance (i.e., non-projected separation):

```
>>> c1 = ICRSCoordinates('5h23m34.5s -69d45m22s', distance=Distance(49, u.kpc))
>>> c2 = ICRSCoordinates('0h52m44.8s -72d49m43s', distance=Distance(61, u.kpc))
>>> sep3d = c1.separation_3d(c2)
>>> sep3d
<Distance 23.05685 kpc>
>>> sep3d.kpc
23.05684814695706
>>> sep3d.Mpc
0.02305684814695706
>>> sep3d.au
4755816315.663559
```

Transforming Between Systems

`astropy.coordinates` supports a rich system for transforming coordinates from one system to another. The key concept is that a registry of all the transformations is used to determine which coordinates can convert to others. When you ask for a transformation, the registry (or "transformation graph") is searched for the shortest path from your starting coordinate to your target, and it applies all of the transformations in that path in series. This allows only the simplest transformations to be defined, and the package will automatically determine how to combine those transformations to get from one system to another.

As described above, there are two ways of transforming coordinates. Coordinates that have an alias (created with `coordinate_alias`) can be converted by simply using attribute style access to any other coordinate system:

```
>>> gc = GalacticCoordinates(l=0, b=45, unit=(u.degree, u.degree))
>>> gc.fk5
<FK5Coordinates RA=229.27250 deg, Dec=-1.12842 deg>
>>> ic = ICRSCoordinates(ra=0, dec=45, unit=(u.degree, u.degree)))
>>> ic.fk5
<FK5Coordinates RA=0.00001 deg, Dec=45.00000 deg>
```

While this appears to be simple attribute-style access, it is actually just syntactic sugar for the `transform_to` method:

```
>>> from astropy.coordinates import FK5Coordinates
>>> gc.transform_to(FK5Coordinates)
<FK5Coordinates RA=229.27250 deg, Dec=-1.12842 deg>
>>> ic.transform_to(FK5Coordinates)
<FK5Coordinates RA=0.00001 deg, Dec=45.00000 deg>
```

The full list of supported coordinate systems and transformations is in the `astropy.coordinates` API documentation below.

Additionally, some coordinate systems support precessing the coordinate to produce a new coordinate in the same system but at a different equinox. Note that these systems have a default equinox they start with if you don't specify one:

```
>>> fk5c = FK5Coordinates('02h31m49.09s +89d15m50.8s')
>>> fk5c.equinox
<Time object: scale='utc' format='jyear_str' vals=J2000.000>
>>> fk5c
<FK5Coordinates RA=37.95454 deg, Dec=89.26411 deg>
>>> fk5c.precess_to(Time(2100, format='jyear', scale='utc'))
<FK5Coordinates RA=88.32396 deg, Dec=89.54057 deg>
```

You can also specify the equinox when you create a coordinate using an `astropy.time.Time` object:

```
>>> from astropy.time import Time
>>> fk5c = FK5Coordinates('02h31m49.09s +89d15m50.8s', equinox=Time('J1970', scale='utc'))
<FK5Coordinates RA=37.95454 deg, Dec=89.26411 deg>
>>> fk5c.precess_to(Time(2000, format='jyear', scale='utc'))
<FK5Coordinates RA=48.02317 deg, Dec=89.38672 deg>
```

Coordinate systems do not necessarily all support an equinox nor precession, as it is a meaningless action for coordinate systems that do not depend on a particular equinox.

Furthermore, coordinates typically have an `obstime` attribute, intended to record the time of the observation. Some systems (especially FK4) require this information due to being non-inertial frames (i.e., they rotate over time due to motions of the defining stars).

Designing Coordinate Systems

New coordinate systems can easily be added by users by simply subclassing the `SphericalCoordinatesBase` object. Detailed instructions for subclassing are in the docstrings for that class. If defining a latitude/longitude style of coordinate system, the `_initialize_latlong` method and `_init_docstring_param_templ` attribute are helpful for automated processing of the inputs.

To define transformations to and from this coordinate, the easiest method is to define a function that accepts an object in one coordinate system and returns the other. Decorate this function with `transform_function` function decorator, supplying the information to determine which coordinates the function transforms to or from. This will register the transformation, allowing any other coordinate object to use this converter. You can also use the `static_transform_matrix` and `dynamic_transform_matrix` decorators to specify the transformation in terms of 3 x 3 cartesian coordinate transformation matrices (typically rotations).

In addition, another resource for the capabilities of this package is the `astropy.coordinates.tests.test_api` 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 looking at it directly if you downloaded a copy of the astropy source code, or typing the following in an IPython session:

```
In [1]: from astropy.coordinates.tests import test_api
In [2]: test_api??
```

1.7.4 See Also

Some references particularly useful in understanding subtleties of the coordinate systems implemented here include:

- **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.
- **USNO Circular 179**
  A useful guide to the IAU 2000/2003 work surrounding ICRS/IERS/CIRS and related problems in precision coordinate system work.
- **Meeus, J. "Astronomical Algorithms"**
  A valuable text describing details of a wide range of coordinate-related problems and concepts.

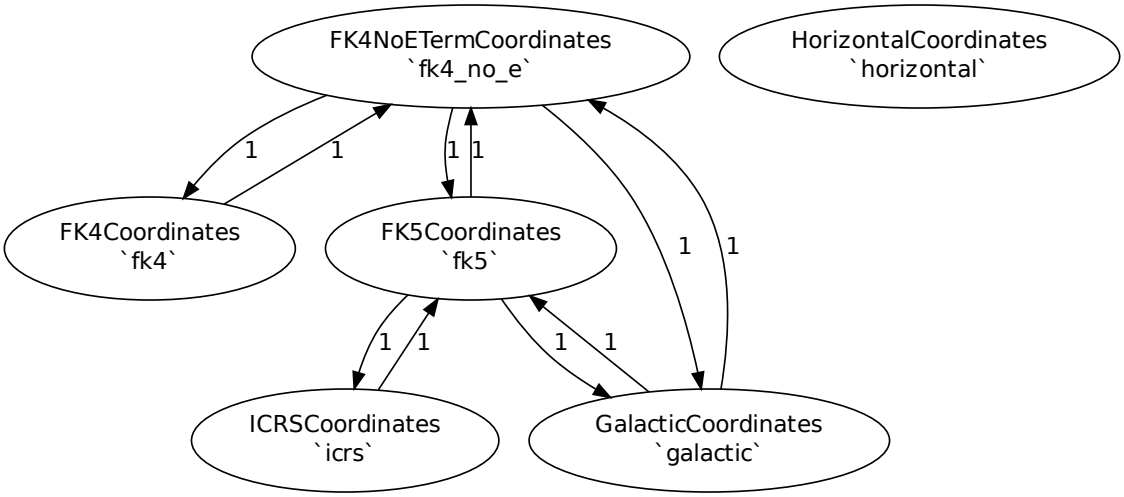
1.7.5 Reference/API

astropy.coordinates Module

This subpackage contains classes and functions for celestial coordinates of astronomical objects. It also contains a framework for conversions between coordinate systems.

The diagram below shows all of the coordinate systems built into the `coordinates` package, their aliases (usable 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 trasnformation between these systems, but the pre-defined transformations should be sufficient for typical usage.

The graph also indicates the priority for each transformation as a number next to the arrow. These priorities are used to decide the preferred order when two trasnformation paths have the same number of steps. These priorities are defined such that path with a *smaller* total priority are favored over larger. E.g., the path from `ICRSCoordinates` to `GalacticCoordinates` goes through `FK5Coordinates` because the total path length is 2 instead of 2.03.



Functions

cartesian_to_spherical(x, y, z)	Converts 3D rectangular cartesian coordinates to spherical polar coordinates.
coordinate_alias(name[, coordcls])	Gives a short name to this coordinate system, allowing other coordinate objects t
dynamic_transform_matrix(fromsys, tosys[, ...])	A function decorator for defining transformations between coordinate systems u
spherical_to_cartesian(r, lat, lon)	Converts spherical polar coordinates to rectangular cartesian coordinates.
static_transform_matrix(fromsys, tosys[, ...])	A function decorator for defining transformations between coordinate systems u
transform_function(fromsys, tosys[, ...])	A function decorator for defining transformations between coordinate systems.

cartesian_to_spherical

astropy.coordinates.distances.cartesian_to_spherical(*x*, *y*, *z*)

Converts 3D rectangular cartesian coordinates to spherical polar coordinates.

Note that the resulting angles are latitude/longitude or elevation/azimuthal form. I.e., the origin is along the equator rather than at the north pole.

Note: This is a low-level function used internally in astropy.coordinates. It is provided for users if they really want to use it, but it is recommended that you use the astropy.coordinates coordinate systems.

Parameters

x : scalar or array-like

The first cartesian coordinate.

y : scalar or array-like

The second cartesian coordinate.

z : scalar or array-like

The third cartesian coordinate.

Returns

r : float or array

The radial coordinate (in the same units as the inputs).

lat : float or array

The latitude in radians

lon : float or array

The longitude in radians

coordinate_alias

`astropy.coordinates.transformations.coordinate_alias(`*name*, *coordcls=None*`)`

Gives a short name to this coordinate system, allowing other coordinate objects to convert to this one using attribute-style access.

Parameters

name : str

The short alias to use for this coordinate class. Should be a valid python identifier.

coordcls : class or None

Either the coordinate class to register or None to use this as a decorator.

Examples

For use with a class already defined, do:

```
coordinate_alias('fancycoords', MyFancyCoordinateClass)
```

To use as a decorator, do:

```
@coordiante_alias('fancycoords')
class MyFancyCoordinateClass(SphericalCoordinatesBase):
    ...
```

dynamic_transform_matrix

`astropy.coordinates.transformations.dynamic_transform_matrix(`*fromsys*, *tosys*, *priority=1*`)`

A function decorator for defining transformations between coordinate systems using a function that yields a matrix.

The decorated function should accept a single argument, the coordinate object to be transformed, and should return a 3 x 3 matrix.

Note: If decorating a static method of a class, `@staticmethod` should be added *above* this decorator.

Parameters

fromsys : class

The coordinate system this function starts from.

tosys : class

The coordinate system this function results in.

priority : number

The priority if this transform when finding the shortest coordinate tranform path - large numbers are lower priorities.

spherical_to_cartesian
`astropy.coordinates.distances.spherical_to_cartesian(`*r*, *lat*, *lon*`)`
Converts spherical polar coordinates to rectangular cartesian coordinates.

Note that the input angles should be in latitude/longitude or elevation/azimuthal form. I.e., the origin is along the equator rather than at the north pole.

Note: This is a low-level function used internally in `astropy.coordinates`. It is provided for users if they really want to use it, but it is recommended that you use the `astropy.coordinates` coordinate systems.

Parameters

r : scalar or array-like

The radial coordinate (in the same units as the inputs).

lat : scalar or array-like

The latitude in radians

lon : scalar or array-like

The longitude in radians

Returns

x : float or array

The first cartesian coordinate.

y : float or array

The second cartesian coordinate.

z : float or array

The third cartesian coordinate.

static_transform_matrix
`astropy.coordinates.transformations.static_transform_matrix(`*fromsys*, *tosys*, *priority=1*`)`
A function decorator for defining transformations between coordinate systems using a matrix.

The decorated function should accept *no* arguments and yield a 3 x 3 matrix.

Note: If decorating a static method of a class, `@staticmethod` should be added *above* this decorator.

Parameters

fromsys : class

The coordinate system this function starts from.

tosys : class

The coordinate system this function results in.

priority : number

The priority if this transform when finding the shortest coordinate tranform path - large numbers are lower priorities.

transform_function
`astropy.coordinates.transformations.transform_function`(*fromsys*, *tosys*, *copyobstime=True*, *priority=1*)

A function decorator for defining transformations between coordinate systems.

Note: If decorating a static method of a class, `@staticmethod` should be added *above* this decorator.

Parameters

fromsys : class

The coordinate system this function starts from.

tosys : class

The coordinate system this function results in.

copyobstime : bool

If True (default) the value of the `_obstime` attribute will be copied to the newly-produced coordinate.

priority : number

The priority if this transform when finding the shortest coordinate tranform path - large numbers are lower priorities.

Classes

Angle(angle[, unit, bounds])	An angle.
AngularSeparation(lat1, lon1, lat2, lon2, units)	An on-sky separation between two directions.
BoundsError	Raised when an angle is outside of its user-specified bounds.
CartesianPoints(x, y, z[, unit])	A cartesian representation of a point in three-dimensional space.
CompositeStaticMatrixTransform(fromsys, ...)	A MatrixTransform constructed by combining a sequence of matricies together
ConvertError	Raised if a coordinate system cannot be converted to another
Dec(angle[, unit])	Represents a declination value.
Distance(*args, **kwargs)	A one-dimensional distance.
DynamicMatrixTransform(fromsys, tosys, ...)	A coordinate transformation specified as a function that yields a 3 x 3 cartesian t
FK4Coordinates(*args, **kwargs)	A coordinate in the FK4 system.
FK4NoETermCoordinates(*args, **kwargs)	A coordinate in the FK4 system.
FK5Coordinates(*args, **kwargs)	A coordinate in the FK5 system.
FunctionTransform(fromsys, tosys, func[, ...])	A coordinate transformation defined by a function that simply accepts a coordina
GalacticCoordinates(*args, **kwargs)	A coordinate in Galactic Coordinates.
HorizontalCoordinates(*args, **kwargs)	A coordinate in the Horizontal or "az/el" system.
ICRSCoordinates(*args, **kwargs)	A coordinate in the ICRS.
IllegalHourError(hour)	Raised when an hour value is not in the range [0,24).
IllegalMinuteError(minute)	Raised when a minute value is not in the range [0,60).
IllegalSecondError(second)	Raised when an second value (time) is not in the range [0,60).
RA(angle[, unit])	An object that represents a right ascension angle.
RangeError	Raised when some part of an angle is out of its valid range.

Table 1.69 – continued from previous page

SphericalCoordinatesBase(*args, **kwargs)	Abstract superclass for all coordinate classes representing points in three dimens
StaticMatrixTransform(fromsys, tosys, matrix)	A coordinate transformation defined as a 3 x 3 cartesian transformation matrix.
UnitsError	Raised if units are missing or invalid.

Angle

class astropy.coordinates.angles.Angle(*angle*, *unit=None*, *bounds=(-360*, *360)*)

Bases: object

An angle.

An angle can be specified either as a float, tuple (see below), or string. If A string, it must be in one of the following formats:

- '1:2:3.4'
- '1 2 3.4'
- '1h2m3.4s'
- '1d2m3.4s'

Parameters

angle : float, int, str, tuple

The angle value. If a tuple, will be interpreted as (h, m s) or (d, m, s) depending on `unit`. If a string, it will be interpreted following the rules described above.

unit : UnitBase, str

The unit of the value specified for the angle. This may be any string that `Unit` understands, but it is better to give an actual unit object. Must be one of `degree`, `radian`, or `hour`.

bounds : tuple

A tuple indicating the upper and lower value that the new angle object may have.

Raises

'~astropy.coordinates.errors.UnitsError' :

If a unit is not provided or it is not hour, radian, or degree.

Attributes Summary

hours	The angle's value in hours (read-only property).
hms	The angle's value in hours, and print as an (h,m,s) tuple (read-only property).
radians	The angle's value in radians (read-only property).
bounds	" The angle's bounds, an immutable property.
degrees	The angle's value in degrees (read-only property).
dms	The angle's value in degrees, and print as an (d,m,s) tuple (read-only property).

Methods Summary

format([unit, decimal, sep, precision, ...])	A string representation of the angle.

Attributes Documentation

`hours`
The angle's value in hours (read-only property).

`hms`
The angle's value in hours, and print as an (h,m,s) tuple (read-only property).

`radians`
The angle's value in radians (read-only property).

`bounds`
" The angle's bounds, an immutable property.

`degrees`
The angle's value in degrees (read-only property).

`dms`
The angle's value in degrees, and print as an (d,m,s) tuple (read-only property).

Methods Documentation

`format`(*unit=Unit("deg")*, *decimal=False*, *sep='fromunit'*, *precision=5*, *alwayssign=False*, *pad=False*)
A string representation of the angle.

Parameters
units : `UnitBase`

Specifies the units, should be 'degree', 'hour', or 'radian'

decimal : bool

If True, a decimal respresentation will be used, otherwise the returned string will be in sexagesimal form.

sep : str

The separator between numbers in a sexagesimal representation. E.g., if it is ':', the result is "12:41:11.1241". Also accepts 2 or 3 separators. E.g., `sep='hms'` would give the result "12h41m11.1241s", or sep='-:' would yield "11-21:17.124". Alternatively, the special string 'fromunit' means 'dms' if the unit is degrees, or 'hms' if the unit is hours.

precision : int

The level of decimal precision. if `decimal` is True, this is the raw precision, otherwise it gives the precision of the last place of the sexagesimal representation (seconds).

alwayssign : bool

If True, include the sign no matter what. If False, only include the sign if it is necessary (negative).

pad : bool

If True, include leading zeros when needed to ensure a fixed number of characters for sexagesimal representation.

Returns
strrepr : str

A string representation of the angle.

AngularSeparation
class astropy.coordinates.angles.AngularSeparation(*lat1*, *lon1*, *lat2*, *lon2*, *units*)
Bases: astropy.coordinates.angles.Angle

An on-sky separation between two directions.

Note: This is computed using the Vincenty great circle distance formula, and hence should be numerically stable even for near antipodal points.

Parameters
lat1 : float

The value of the first latitudinal/elevation angle.

lon1 : float

The value of the first longitudinal/azimuthal angle.

lat2 : float

The value of the second latitudinal/elevation angle.

lon2 : float

The value of the second longitudinal/azimuthal angle.

units : units

The units of the given angles.

Attributes Summary

arcsecs	The value of this separation in arcseconds.
arcmins	The value of this separation in arcminutes.

Attributes Documentation

arcsecs
The value of this separation in arcseconds.

arcmins
The value of this separation in arcminutes.

BoundsError
exception astropy.coordinates.errors.BoundsError
Raised when an angle is outside of its user-specified bounds.

CartesianPoints
class astropy.coordinates.distances.CartesianPoints(*x*, *y*, *z*, *unit=None*)
Bases: object

A cartesian representation of a point in three-dimensional space.

Attributes

x	number or array	The first cartesian coordinate.
y	number or array	The second cartesian coordinate.
z	number or array	The third cartesian coordinate.
unit	UnitBase object or None	The physical unit of the coordinate values.

Methods Summary

to_spherical()	Converts to the spherical representation of this point.

Methods Documentation

to_spherical()
Converts to the spherical representation of this point.

Returns

r : float or array

The radial coordinate (in the same units as the inputs).

lat : float or array

The latitude in radians

lon : float or array

The longitude in radians

CompositeStaticMatrixTransform

class astropy.coordinates.transformations.CompositeStaticMatrixTransform(*fromsys*, *tosys*, *matricies*, *priority=1*, *register=True*)

Bases: astropy.coordinates.transformations.StaticMatrixTransform

A MatrixTransform constructed by combining a sequence of matricies together. See MatrixTransform for syntax details.

Parameters

fromsys : class

The coordinate system *class* to start from.

tosys : class

The coordinate system *class* to transform into.

matrices: sequence of array-like :

A sequence of 3 x 3 cartesian transformation matricies.

priority : number

The priority if this transform when finding the shortest coordinate tranform path - large numbers are lower priorities.

ConvertError
exception astropy.coordinates.errors.ConvertError
Raised if a coordinate system cannot be converted to another

Dec
class astropy.coordinates.angles.Dec(*angle*, *unit=None*)
Bases: astropy.coordinates.angles.Angle

Represents a declination value.

This object can be created from a numeric value along with a unit, or else a string in any commonly represented format, e.g. "12 43 23.53", "-32d52m29s". Unless otherwise specified via the 'unit' parameter, degrees are assumed. Bounds are fixed to [-90,90] degrees.

Parameters
angle : float, int, str, tuple

The angle value. If a tuple, will be interpreted as (h, m s) or (d, m, s) depending on `unit`. If a string, it will be interpreted following the rules described above.

unit : UnitBase, str

The unit of the value specified for the angle. This may be any string that `Unit` understands, but it is better to give an actual unit object. Must be one of `degree`, `radian`, or `hour`.

bounds : tuple

A tuple indicating the upper and lower value that the new angle object may have.

Raises
'~astropy.coordinates.errors.UnitsError' :

If a unit is not provided or it is not hour, radian, or degree.

Distance
class astropy.coordinates.distances.Distance(**args*, ***kwargs*)
Bases: object

A one-dimensional distance.

This can be initialized in one of two ways, using either a distance and a unit, or a redshift and (optionally) a cosmology. `value` and `unit` may be provided as positional arguments, but `z` and `cosmology` are only valid as keyword arguments (see examples).

Parameters
value : scalar

The value of this distance

unit : UnitBase

The units for this distance. Must have dimensions of distance.

z : float

A redshift for this distance. It will be converted to a distance by computing the luminosity distance for this redshift given the cosmology specified by `cosmology`.

cosmology : Cosmology or None

A cosmology that will be used to compute the distance from `z`. If None, the current cosmology will be used (see astropy.cosmology for details).

Raises

UnitsError :

If the `unit` is not a distance.

Examples

```
>>> from astropy import units as u
>>> from astropy.cosmology import WMAP3
>>> d1 = Distance(10, u.Mpc)
>>> d2 = Distance(40, unit=u.au)
>>> d3 = Distance(value=5, unit=u.kpc)
>>> d4 = Distance(z=0.23)
>>> d5 = Distance(z=0.23, cosmology=WMAP3)
```

Attributes Summary

`Mpc`	The value of this distance in megaparsecs
`pc`	The value of this distance in parsecs
`m`	The value of this distance in meters
`km`	The value of this distance in kilometers
`kpc`	The value of this distance in kiloparsecs
`au`	The value of this distance in astronomical units
`lightyear`	The value of this distance in light years
`z`	The redshift for this distance assuming its physical distance is a luminosity distance.

Methods Summary

`compute_z`([cosmology])	The redshift for this distance assuming its physical distance is a luminosity distance.

Attributes Documentation

`Mpc`
: The value of this distance in megaparsecs

`pc`
: The value of this distance in parsecs

`m`
: The value of this distance in meters

`km`
: The value of this distance in kilometers

`kpc`
: The value of this distance in kiloparsecs

`au`
: The value of this distance in astronomical units

`lightyear`
: The value of this distance in light years

z

The redshift for this distance assuming its physical distance is a luminosity distance.

Note: This uses the "current" cosmology to determine the appropriate distance to redshift conversions. See `astropy.cosmology` for details on how to change this.

Methods Documentation

compute_z(*cosmology=None*)

The redshift for this distance assuming its physical distance is a luminosity distance.

Parameters

cosmology : cosmology or None

The cosmology to assume for this calculation, or None to use the current cosmology.

DynamicMatrixTransform

class astropy.coordinates.transformations.DynamicMatrixTransform(*fromsys*, *tosys*, *matrix_func*, *priority=1*, *register=True*)

Bases: astropy.coordinates.transformations.CoordinateTransform

A coordinate transformation specified as a function that yields a 3 x 3 cartesian transformation matrix.

Parameters

fromsys : class

The coordinate system *class* to start from.

tosys : class

The coordinate system *class* to transform into.

matrix_func: callable :

A callable that accepts a coordinate object and yields the 3 x 3 matrix that converts it to the new coordinate system.

priority : number

The priority if this transform when finding the shortest coordinate tranform path - large numbers are lower priorities.

Raises

TypeError :

If matrix_func is not callable

FK4Coordinates

class astropy.coordinates.builtin_systems.FK4Coordinates(**args*, ***kwargs*)

Bases: astropy.coordinates.coordsystems.SphericalCoordinatesBase

A coordinate in the FK4 system.

Parameters

coordstr : str

A single string with the coordinates. Cannot be used with dec and ra nor x/y/z.

ra : Angle, float, int, str

This must be given with dec.

dec : Angle, float, int, str

This must be given with ra.

distance : Distance, optional

This may be given with dec and ra or coordstr and not x, y, or z. If not given, None (unit sphere) will be assumed.

x : number

The first cartesian coordinate. Must be given with y and z and not with ra or dec nor coordstr.

y : number

The second cartesian coordinate. Must be given with x and z and not with ra or dec nor coordstr.

z : number

The third cartesian coordinate. Must be given with x and y and not with ra or dec nor coordstr.

cartpoint : CartesianPoints

A cartesian point with the coordinates. Cannot be used with any other arguments.

unit :

The unit parameter's interpretation depends on what other parameters are given:

- **If** ra **and** dec **or** coordstr **are given:**
  unit must be a length-2 sequence specifying the units of ra and dec, respectively. They can be either UnitBase objects or strings that will be converted using Unit. They can also be None to attempt to automatically interpret the units (see Angle for details.) If unit is just None, this will be interpreted the same as (None, None).
- **If** x**,** y**, and** z **are given:**
  unit must be a single unit with dimensions of length

equinox : Time, optional

The equinox for these coordinates. Defaults to B1950.

obstime : Time or None

The time of observation for this coordinate. If None, it will be taken to be the same as the equinox.

Alternatively, a single argument that is any kind of spherical coordinate :

can be provided, and will be converted to 'FK4Coordinates' and used as this :

coordinate. :

Attributes Summary

Continued on next page

Table 1.76 – continued from previous page

equinox
obstime
lonangle
latangle

Methods Summary

precess_to(newequinox)	Precesses the coordinates from their current equinox to a new equinox.

Attributes Documentation

equinox

obstime

lonangle

latangle

Methods Documentation

precess_to(*newequinox*)

Precesses the coordinates from their current equinox to a new equinox.

Parameters

newequinox : Time

The equinox to precess these coordinates to.

Returns

newcoord : FK4Coordinates

The new coordinate

FK4NoETermCoordinates

class astropy.coordinates.builtin_systems.FK4NoETermCoordinates(**args*, ***kwargs*)

Bases: astropy.coordinates.coordsystems.SphericalCoordinatesBase

A coordinate in the FK4 system.

Parameters

coordstr : str

A single string with the coordinates. Cannot be used with dec and ra nor x/y/z.

ra : Angle, float, int, str

This must be given with dec.

dec : Angle, float, int, str

This must be given with ra.

distance : Distance, optional

This may be given with dec and ra or coordstr and not x, y, or z. If not given, None (unit sphere) will be assumed.

x : number

The first cartesian coordinate. Must be given with y and z and not with ra or dec nor coordstr.

y : number

The second cartesian coordinate. Must be given with x and z and not with ra or dec nor coordstr.

z : number

The third cartesian coordinate. Must be given with x and y and not with ra or dec nor coordstr.

cartpoint : CartesianPoints

A cartesian point with the coordinates. Cannot be used with any other arguments.

unit :

The unit parameter's interpretation depends on what other parameters are given:

- **If** ra **and** dec **or** coordstr **are given:**
  unit must be a length-2 sequence specifying the units of ra and dec, respectively. They can be either UnitBase objects or strings that will be converted using Unit. They can also be None to attempt to automatically interpret the units (see Angle for details.) If unit is just None, this will be interpreted the same as (None, None).
- **If** x**,** y**, and** z **are given:**
  unit must be a single unit with dimensions of length

equinox : Time, optional

The equinox for these coordinates. Defaults to B1950.

obstime : Time or None

The time of observation for this coordinate. If None, it will be taken to be the same as the equinox.

Alternatively, a single argument that is any kind of spherical coordinate :

can be provided, and will be converted to 'FK4NoETermCoordinates' and used as this :

coordinate. :

Attributes Summary

equinox
obstime
lonangle
latangle

Methods Summary

precess_to(newequinox)	Precesses the coordinates from their current equinox to a new equinox.

Attributes Documentation

equinox

obstime

lonangle

latangle

Methods Documentation

precess_to(*newequinox*)

Precesses the coordinates from their current equinox to a new equinox.

Parameters

newequinox : Time

The equinox to precess these coordinates to.

Returns

newcoord : FK4NoETermCoordinates

The new coordinate

FK5Coordinates

class astropy.coordinates.builtin_systems.FK5Coordinates(**args*, ***kwargs*)

Bases: astropy.coordinates.coordsystems.SphericalCoordinatesBase

A coordinate in the FK5 system.

Parameters

coordstr : str

A single string with the coordinates. Cannot be used with dec and ra nor x/y/z.

ra : Angle, float, int, str

This must be given with dec.

dec : Angle, float, int, str

This must be given with ra.

distance : Distance, optional

This may be given with dec and ra or coordstr and not x, y, or z. If not given, None (unit sphere) will be assumed.

x : number

The first cartesian coordinate. Must be given with `y` and `z` and not with `ra` or `dec` nor `coordstr`.

y : number

The second cartesian coordinate. Must be given with `x` and `z` and not with `ra` or `dec` nor `coordstr`.

z : number

The third cartesian coordinate. Must be given with `x` and `y` and not with `ra` or `dec` nor `coordstr`.

cartpoint : `CartesianPoints`

A cartesian point with the coordinates. Cannot be used with any other arguments.

unit :

The `unit` parameter's interpretation depends on what other parameters are given:

- **If `ra` and `dec` or `coordstr` are given:**
  `unit` must be a length-2 sequence specifying the units of `ra` and `dec`, respectively. They can be either `UnitBase` objects or strings that will be converted using `Unit`. They can also be None to attempt to automatically interpret the units (see `Angle` for details.) If `unit` is just `None`, this will be interpreted the same as `(None, None)`.
- **If `x`, `y`, and `z` are given:**
  `unit` must be a single unit with dimensions of length

equinox : `Time`, optional

The equinox for these coordinates. Defaults to J2000.

obstime : `Time` or None

The time of observation for this coordinate. If None, it will be taken to be the same as the `equinox`.

Alternatively, a single argument that is any kind of spherical coordinate :

can be provided, and will be converted to 'FK5Coordinates' and used as this :

coordinate. :

Attributes Summary

`equinox`
`obstime`
`lonangle`
`latangle`

Methods Summary

`precess_to`(newequinox)	Precesses the coordinates from their current `equinox` to a new equinox and returns the resulting coordina

Attributes Documentation

`equinox`

`obstime`

`lonangle`

`latangle`

Methods Documentation

`precess_to`(*newequinox*)

Precesses the coordinates from their current `equinox` to a new equinox and returns the resulting coordinate.

Parameters

newequinox : `Time`

The equinox to precess these coordinates to.

Returns

newcoord : FK5Coordinates

The new coordinate

FunctionTransform

class `astropy.coordinates.transformations.FunctionTransform`(*fromsys*, *tosys*, *func*, *copyobstime=True*, *priority=1*, *register=True*)

Bases: `astropy.coordinates.transformations.CoordinateTransform`

A coordinate transformation defined by a function that simply accepts a coordinate object and returns the transformed coordinate object.

Parameters

fromsys : class

The coordinate system *class* to start from.

tosys : class

The coordinate system *class* to transform into.

func : callable

The transformation function.

copyobstime : bool

If True (default) the value of the `_obstime` attribute will be copied to the newly-produced coordinate.

priority : number

The priority if this transform when finding the shortest coordinate tranform path - large numbers are lower priorities.

register : bool

Determines if this transformation will be registered in the astropy master transform graph.

Raises

TypeError :

If `func` is not callable.

ValueError :

If `func` cannot accept one argument.

GalacticCoordinates

class `astropy.coordinates.builtin_systems.GalacticCoordinates`(**args*, ***kwargs*)

Bases: `astropy.coordinates.coordsystems.SphericalCoordinatesBase`

A coordinate in Galactic Coordinates.

Note: Transformations from Galactic Coordinates to other systems are not well-defined because of ambiguities in the definition of Galactic Coordinates. See `Lie et al. 2011` for more details on this. Here, we use the `Reid & Brunthaler 2004` definition for converting to/from FK5, and assume the IAU definition applies for converting to FK4 *without* e-terms.

Parameters

coordstr : str

A single string with the coordinates. Cannot be used with `b` and `l` nor `x/y/z`.

l : `Angle`, float, int, str

This must be given with `b`.

b : `Angle`, float, int, str

This must be given with `l`.

distance : `Distance`, optional

This may be given with `b` and `l` or `coordstr` and not `x`, `y`, or `z`. If not given, `None` (unit sphere) will be assumed.

x : number

The first cartesian coordinate. Must be given with `y` and `z` and not with `l` or `b` nor `coordstr`.

y : number

The second cartesian coordinate. Must be given with `x` and `z` and not with `l` or `b` nor `coordstr`.

z : number

The third cartesian coordinate. Must be given with `x` and `y` and not with `l` or `b` nor `coordstr`.

cartpoint : `CartesianPoints`

A cartesian point with the coordinates. Cannot be used with any other arguments.

unit :

The `unit` parameter's interpretation depends on what other parameters are given:

- **If `l` and `b` or `coordstr` are given:**
  `unit` must be a length-2 sequence specifying the units of `l` and `b`, respectively. They can be either `UnitBase` objects or strings that will be converted using `Unit`. They can also be None to attempt to automatically interpret the units (see `Angle` for details.) If `unit` is just `None`, this will be interpreted the same as `(None, None)`.
- **If `x`, `y`, and `z` are given:**
  `unit` must be a single unit with dimensions of length

obstime : `Time` or None

The time of observation for this coordinate. If None, it will be taken to be the same as the `equinox`.

Alternatively, a single argument that is any kind of spherical coordinate :

can be provided, and will be converted to 'GalacticCoordinates' and :

used as this coordinate. :

Attributes Summary

`lonangle`
`latangle`

Attributes Documentation

`lonangle`

`latangle`

HorizontalCoordinates

class `astropy.coordinates.builtin_systems.HorizontalCoordinates(*args, **kwargs)`

Bases: `astropy.coordinates.coordsystems.SphericalCoordinatesBase`

A coordinate in the Horizontal or "az/el" system.

Parameters

coordstr : str

A single string with the coordinates. Cannot be used with `el` and `az` nor `x`/`y`/`z`.

az : `Angle`, float, int, str

This must be given with `el`.

el : `Angle`, float, int, str

This must be given with `az`.

distance : `Distance`, optional

This may be given with `el` and `az` or `coordstr` and not `x`, `y`, or `z`. If not given, `None` (unit sphere) will be assumed.

x : number

The first cartesian coordinate. Must be given with `y` and `z` and not with `az` or `el` nor `coordstr`.

y : number

The second cartesian coordinate. Must be given with `x` and `z` and not with `az` or `el` nor `coordstr`.

z : number

The third cartesian coordinate. Must be given with `x` and `y` and not with `az` or `el` nor `coordstr`.

cartpoint : `CartesianPoints`

A cartesian point with the coordinates. Cannot be used with any other arguments.

unit :

The `unit` parameter's interpretation depends on what other parameters are given:

- **If `az` and `el` or `coordstr` are given:**
  `unit` must be a length-2 sequence specifying the units of `az` and `el`, respectively. They can be either `UnitBase` objects or strings that will be converted using `Unit`. They can also be None to attempt to automatically interpret the units (see `Angle` for details.) If `unit` is just `None`, this will be interpreted the same as `(None, None)`.
- **If `x`, `y`, and `z` are given:**
  `unit` must be a single unit with dimensions of length

equinox : `Time`, optional

The equinox for these coordinates. Defaults to J2000.

obstime : `Time` or None

The time of observation for this coordinate. If None, it will be taken to be the same as the `equinox`.

Alternatively, a single argument that is any kind of spherical coordinate :

can be provided, and will be converted to 'HorizontalCoordinates' and used :

as this coordinate. :

Attributes Summary

`equinox`
`obstime`
`lonangle`
`latangle`

Attributes Documentation

`equinox`

`obstime`

lonangle

latangle

ICRSCoordinates

class astropy.coordinates.builtin_systems.ICRSCoordinates(**args*, ***kwargs*)

Bases: astropy.coordinates.coordsystems.SphericalCoordinatesBase

A coordinate in the ICRS.

If you're looking for "J2000" coordinates, and aren't sure if you want to use this or FK5Coordinates, you probably want to use ICRS. It's more well-defined as a catalog coordinate and is an inertial system.

Parameters

coordstr : str

A single string with the coordinates. Cannot be used with dec and ra nor x/y/z.

ra : Angle, float, int, str

This must be given with dec.

dec : Angle, float, int, str

This must be given with ra.

distance : Distance, optional

This may be given with dec and ra or coordstr and not x, y, or z. If not given, None (unit sphere) will be assumed.

x : number

The first cartesian coordinate. Must be given with y and z and not with ra or dec nor coordstr.

y : number

The second cartesian coordinate. Must be given with x and z and not with ra or dec nor coordstr.

z : number

The third cartesian coordinate. Must be given with x and y and not with ra or dec nor coordstr.

cartpoint : CartesianPoints

A cartesian point with the coordinates. Cannot be used with any other arguments.

unit :

The unit parameter's interpretation depends on what other parameters are given:

- **If** ra **and** dec **or** coordstr **are given:**
  unit must be a length-2 sequence specifying the units of ra and dec, respectively. They can be either UnitBase objects or strings that will be converted using Unit. They can also be None to attempt to automatically interpret the units (see Angle for details.) If unit is just None, this will be interpreted the same as (None, None).

- **If** `x`**,** `y`**, and** `z` **are given:**
  `unit` must be a single unit with dimensions of length

obstime : `Time` or None

The time of observation for this coordinate. If None, it will be taken to be the same as the `equinox`.

Alternatively, a single argument that is any kind of spherical coordinate :

can be provided, and will be converted to ICRSCoordinates and used as this :

coordinate. :

Attributes Summary

`equinox`
`obstime`
`lonangle`
`latangle`

Attributes Documentation

`equinox`

`obstime`

`lonangle`

`latangle`

IllegalHourError

exception `astropy.coordinates.errors.IllegalHourError`(*hour*)

Raised when an hour value is not in the range [0,24).

Usage:

```
if not 0 <= hr < 24:
raise IllegalHourError(hour)
```

Parameters

hour : int, float

IllegalMinuteError

exception `astropy.coordinates.errors.IllegalMinuteError`(*minute*)

Raised when an minute value is not in the range [0,60).

Usage:

if not 0 <= min < 60:
raise IllegalMinuteError(minute)

Parameters
minute : int, float

IllegalSecondError
exception `astropy.coordinates.errors.IllegalSecondError`(*second*)
Raised when an second value (time) is not in the range [0,60).

Usage:

if not 0 <= sec < 60:
raise IllegalSecondError(second)

Parameters
second : int, float

RA
class `astropy.coordinates.angles.RA`(*angle*, *unit=None*)
Bases: `astropy.coordinates.angles.Angle`

An object that represents a right ascension angle.

This object can be created from a numeric value along with a unit. If the value specified is greater than "24", then a unit of degrees is assumed. Bounds are fixed to [0,360] degrees.

Parameters
angle : float, int, str, tuple

The angle value. If a tuple, will be interpreted as (h, m s) or (d, m, s) depending on `unit`. If a string, it will be interpreted following the rules described above.

unit : `UnitBase`, str

The unit of the value specified for the angle. This may be any string that `Unit` understands, but it is better to give an actual unit object. Must be one of `degree`, `radian`, or `hour`.

Raises
'~astropy.coordinates.errors.UnitsError' :

If a unit is not provided or it is not hour, radian, or degree.

Methods Summary

`hour_angle`(lst)	Computes the hour angle for this RA given a local sidereal time (LST).
`lst`(hour_angle)	Calculates the local sidereal time (LST) if this RA is at a particular hour angle.

Methods Documentation

`hour_angle`(*lst*)
Computes the hour angle for this RA given a local sidereal time (LST).

Parameters
lst : `Angle`, `Time`

A local sidereal time (LST).

Returns
hour_angle : Angle

The hour angle for this RA at the LST lst.

lst(*hour_angle*)
Calculates the local sidereal time (LST) if this RA is at a particular hour angle.

Parameters
hour_angle : Angle

An hour angle.

Returns
lst : Angle

The local siderial time as an angle.

RangeError
exception astropy.coordinates.errors.RangeError
Raised when some part of an angle is out of its valid range.

SphericalCoordinatesBase
class astropy.coordinates.coordsystems.SphericalCoordinatesBase(**args*, ***kwargs*)
Bases: object

Abstract superclass for all coordinate classes representing points in three dimensions.

Notes

Subclasses must implement __init__, and define the latangle and lonangle properties. They may also override the equinox property, or leave it unaltered to indicate the coordinates are equinoxless.

_initialize_latlon is provided to implement typical initialization features, and should be called from a subclass' __init__. See the classes in astropy.coordinates.builtin_systems for examples of this.

Attributes Summary

y	
lonangle	The longitudinal/azimuthal angle for these coordinates as an Angle object.
equinox	The equinox of this system, or None to indicate no equinox specified.
distance	The radial distance for this coordinate object as an Distance object.
cartesian	
latangle	The latitudinal/elevation angle for these coordinates as an Angle object.
x	
z	

Methods Summary

separation(other)	Computes on-sky separation between this coordinate and another.
is_transformable_to(tosys)	Determines if this coordinate can be transformed to a particular system.

Continued on next page

Table 1.87 – continued from previous page

separation_3d(other)	Computes three dimensional separation between this coordinate and another.
transform_to(tosys)	Transform this coordinate to a new system.

Attributes Documentation

y

lonangle
The longitudinal/azimuthal angle for these coordinates as an Angle object.

Note: This should be overridden in subclasses as a read-only property that just returns an attribute a way to abstract the exact choice of names for the coordinates. E.g., ICRSCoordinates implements this by doing return self.dec.

---

equinox
The equinox of this system, or None to indicate no equinox specified.

distance
The radial distance for this coordinate object as an Distance object.

If set as a tuple, the tuple will be passed into the Distance constructor.

Alternatively, this may be None, indicating an unknown/not given distance. Where necessary, this object will be interpreted as angles on the unit sphere.

cartesian

latangle
The latitudinal/elevation angle for these coordinates as an Angle object.

Note: This should be overridden in subclasses as a read-only property that just returns an attribute a way to abstract the exact choice of names for the coordiantes. E.g., ICRSCoordinates implements this by doing return self.ra.

---

x

z

Methods Documentation

separation(*other*)
Computes on-sky separation between this coordinate and another.

See the AngularSeparation docstring for further details on the actual calculation.

Parameters
other : SphericalCoordinatesBase

The coordinate system to get the separation to.

Returns
sep : AngularSeparation

The on-sky separation between this and the other coordinate.

is_transformable_to(*tosys*)
Determines if this coordinate can be transformed to a particular system.

Parameters
tosys : class

The system to transform this coordinate into.

Returns
transformable : bool or str

True if this can be trasnformed to tosys, False if not. The string 'same' if tosys is the same system as this object (i.e. no transformation is needed).

separation_3d(*other*)
Computes three dimensional separation between this coordinate and another.

Parameters
other : SphericalCoordinatesBase

The coordinate system to get the distance to.

Returns
sep : Distance

The real-space distance between these two coordinates.

Raises
ValueError :

If this or the other coordinate do not have distances.

transform_to(*tosys*)
Transform this coordinate to a new system.

Parameters
tosys : class

The system to transform this coordinate into.

Returns
transcoord :

A new object with this coordinate represented in the tosys system.

Raises
ValueError :

If there is no possible transformation route.

StaticMatrixTransform
class astropy.coordinates.transformations.StaticMatrixTransform(*fromsys*, *tosys*, *matrix*, *priority=1*, *register=True*)
Bases: astropy.coordinates.transformations.CoordinateTransform

A coordinate transformation defined as a 3 x 3 cartesian transformation matrix.

Parameters
fromsys : class

The coordinate system *class* to start from.

tosys : class

The coordinate system *class* to transform into.

matrix: array-like :

A 3 x 3 matrix for transforming 3-vectors. In most cases will be unitary (although this is not strictly required).

priority : number

The priority if this transform when finding the shortest coordinate tranform path - large numbers are lower priorities.

Raises

ValueError :

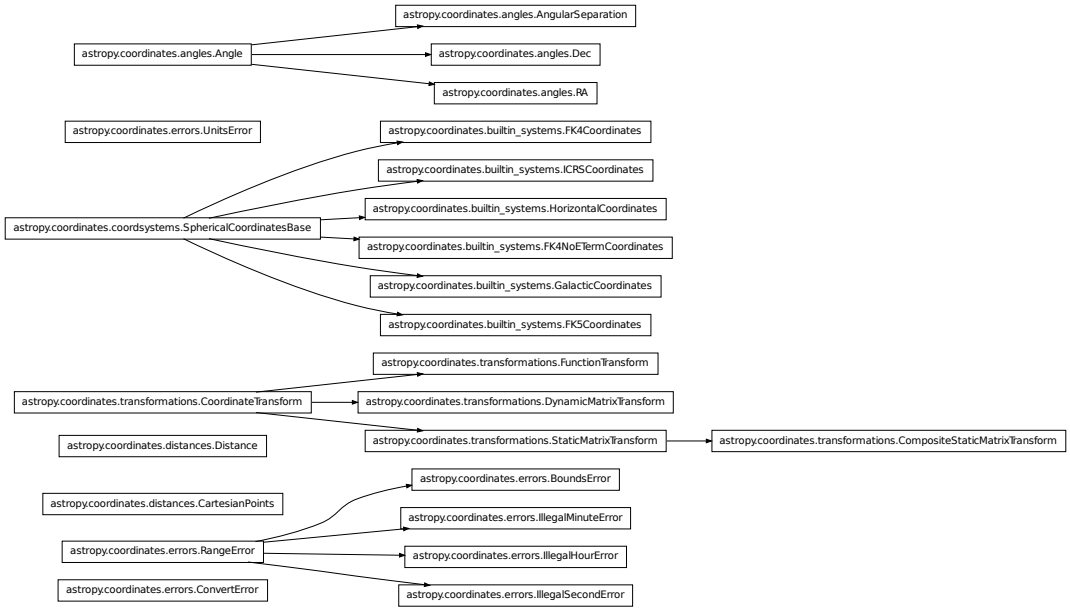
If the matrix is not 3 x 3

UnitsError

exception `astropy.coordinates.errors.UnitsError`

Raised if units are missing or invalid.

Class Inheritance Diagram



1.8 Data Tables (`astropy.table`)

1.8.1 Introduction

`astropy.table` provides functionality for storing and manipulating heterogenous tables of data in a way that is familiar to `numpy` users. A few notable features 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.
- Full support for multidimensional columns.
- Create a table by referencing (not copying) an existing `numpy` table.

Currently `astropy.table` is used when reading an ASCII table using `astropy.io.ascii`. Future releases of AstroPy are expected to use the `Table` class for other subpackages such as `astropy.io.votable` and `astropy.io.fits`.

1.8.2 Getting Started

The basic workflow for creating a table, accessing table elements, and modifying the table is shown below. These examples show a very simple case, while the full `astropy.table` documentation is available from the *Using table* section.

First create a simple table with three columns of data named a, b, and c. These columns have integer, float, and string values respectively:

```
>>> from astropy.table import Table, Column
>>> a = [1, 4, 5]
>>> b = [2.0, 5.0, 8.2]
>>> c = ['x', 'y', 'z']
>>> t = Table([a, b, c], names=('a', 'b', 'c'), meta={'name': 'first table'})
```

There are a few ways to examine the table. You can get detailed information about the table values and column definitions as follows:

```
>>> t
<Table rows=3 names=('a','b','c')>
array([(1, 2.0, 'x'), (4, 5.0, 'y'), (5, 8.2, 'z')],
      dtype=[('a', '<i8'), ('b', '<f8'), ('c', '|S1')])
```

From within the IPython notebook, the table is displayed as a formatted HTML table:

```
In [19]: t
```

Out[19]:

a	b	c
1	2.0	x
4	5.0	y
5	8.2	z

If you print the table (either from the noteboook or in a text console session) then a formatted version appears:

```
>>> print(t)
  a   b   c
--- --- ---
  1 2.0   x
  4 5.0   y
  5 8.2   z
```

For a long table you can scroll up and down through the table one page at time:

```
>>> t.more()
```

Now examine some high-level information about the table:

```
>>> t.colnames
['a', 'b', 'c']
>>> 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' units=None format=None description=None>
array([1, 4, 5])

>>> t['a'][1]    # Row 1 of column 'a'
4

>>> t[1]         # Row obj for with row 1 values
<Row 1 of table
 values=(4, 5.0, 'y')
 dtype=[('a', '<i8'), ('b', '<f8'), ('c', '|S1')]>

>>> t[1]['a']    # Column 'a' of row 1
4
```

One can retreive a subset of a table by rows (using a slice) or 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
--- --- ---
  1 2.0   x
  4 5.0   y

>>> t['a', 'c']  # Table with cols 'a', 'c'
 a   c
--- ---
  1   x
  4   y
  5   z
```

Modifying table values in place is flexible and works as one would expect:

```
>>> t['a'] = [-1, -2, -3]       # Set all column values
>>> t['a'][2] = 30              # Set row 2 of column 'a'
>>> t[1] = (8, 9.0, "W")        # Set all row values
>>> t[1]['b'] = -9              # Set column 'b' of row 1
>>> t[0:2]['b'] = 100.0         # Set column 'c' of rows 0 and 1
>>> print(t)
 a    b    c
--- ----- ---
 -1 100.0   x
  8 100.0   W
 30   8.2   z
```

Add, remove, and rename columns with the following:

```
>>> t.add_column(Column('d', [1, 2, 3]))
>>> t.remove_column('c')
>>> t.rename_column('a', 'A')
>>> t.colnames
['A', 'b', 'd']
```

Adding a new row of data to the table is as follows:

```
>>> t.add_row([-8, -9, 10])
>>> len(t)
4
```

Lastly, one can create a table with support for missing values, for example by setting `masked=True`:

```
>>> t = Table([a, b, c], names=('a', 'b', 'c'), masked=True)
>>> t['a'].mask = [True, True, False]
>>> t
<Table rows=3 names=('a','b','c')>
masked_array(data = [(--, 2.0, 'x') (--, 5.0, 'y') (5, 8.2, 'z')],
             mask = [(True, False, False) (True, False, False) (False, False, False)],
       fill_value = (999999, 1e+20, 'N'),
            dtype = [('a', '<i8'), ('b', '<f8'), ('c', '|S1')])

>>> print(t)
 a   b   c
--- --- ---
 -- 2.0   x
 -- 5.0   y
  5 8.2   z
```

1.8.3 Using `table`

The details of using `astropy.table` are provided in the following sections:

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` constructor are in the Initialization Details section. However, the easiest 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, 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 `Table` and `Column` classes along with the `numpy` package:

```
>>> from astropy.table import Table, Column
>>> 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 columns or rows requires making a new copy of the entire table table each time, so in the case of large tables this may be slow.

```
>>> t = Table()
>>> t.add_column(Column('a', [1, 4]))
>>> t.add_column(Column('b', [2.0, 5.0]))
>>> t.add_column(Column('c', ['x', 'y']))

>>> t = Table(names=('a', 'b', 'c'), dtypes=('f4', 'i4', 'S2'))
>>> t.add_row((1, 2.0, 'x'))
>>> t.add_row((4, 5.0, 'y'))
```

List input 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 auto-generated as col<N>.

```
>>> a = [1, 4]
>>> b = [2.0, 5.0]
>>> c = ['x', 'y']
>>> t = Table([a, b, c], names=('a', 'b', 'c'))
>>> t
<Table rows=2 names=('a','b','c')>
array([(1, 2.0, 'x'), (4, 5.0, 'y')],
      dtype=[('a', '<i8'), ('b', '<f8'), ('c', '|S1')])
```

Make a new table using columns from the first table

Once you have a Table then you can make new table by selecting columns and putting this into a Python list, e.g. [t['c'], t['a']]:

```
>>> Table([t['c'], t['a']])
<Table rows=2 names=('c','a')>
array([('x', 1), ('y', 4)],
      dtype=[('c', '|S1'), ('a', '<i8')])
```

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 rows=2 names=('a','b')>
array([(1, 12.0), (16, 15.0)],
      dtype=[('a', '<i8'), ('b', '<f8')])
```

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('axis', ['x', 'y'])
>>> arr = (a, b, c)
>>> Table(arr)  # Data column named "c" has a name "axis" that table
<Table rows=2 names=('col0','col1','axis')>
array([(1, [2, 3], 'x'), (4, [5, 6], 'y')],
      dtype=[('col0', '<i8'), ('col1', '<i8', (2,)), ('axis', '|S1')])
```

Notice that in the third column the existing column name 'axis' is used.

Dictionary input A dictionary of column data can be used to initialize a Table.

```
>>> arr = {'a': [1, 4],
...        'b': [2.0, 5.0],
...        'c': ['x', 'y']}
>>>
>>> Table(arr)
<Table rows=2 names=('a','c','b')>
array([(1, 'x', 2.0), (4, 'y', 5.0)],
      dtype=[('a', '<i8'), ('c', '|S1'), ('b', '<f8')])
```

Specify the column order and optionally the data types

```
>>> Table(arr, names=('a', 'b', 'c'), dtypes=('f4', 'i4', 'S2'))
<Table rows=2 names=('a','b','c')>
array([(1.0, 2, 'x'), (4.0, 5, 'y')],
      dtype=[('a', '<f4'), ('b', '<i4'), ('c', '|S2')])
```

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('axis', ['x', 'y'])}
>>> Table(arr, names=('a', 'b', 'c'))
<Table rows=2 names=('a','b','c')>
array([(1, [2, 3], 'x'), (4, [5, 6], 'y')],
      dtype=[('a', '<i8'), ('b', '<i8', (2,)), ('c', '|S1')])
```

Notice that the key 'c' takes precendence 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 2-element array.

Renaming columns is not possible

```
>>> Table(arr, names=('a_new', 'b_new', 'c_new'))
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "astropy/table/table.py", line 404, in __init__
    init_func(data, names, dtypes, n_cols, copy)
  File "astropy/table/table.py", line 467, in _init_from_dict
    data_list = [data[name] for name in names]
KeyError: 'a_new'
```

NumPy structured array The structured array is the standard mechanism in numpy for storing heterogenous table data. Most scientific I/O packages that read table files (e.g. PyFITS, vo.table, 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', 'i8'), ('b', 'f8'), ('c', 'S2')])
```

From `arr` it is simple to create the corresponding `Table` object:

```
>>> Table(arr)
<Table rows=2 names=('a','b','c')>
array([(1, 2.0, 'x'), (4, 5.0, 'y')],
      dtype=[('a', '<i8'), ('b', '<f8'), ('c', '|S2')])
```

Note that in the above example and most the following ones 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
```

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 rows=2 names=('a_new','b_new','c_new')>
array([(1, 2.0, 'x'), (4, 5.0, 'y')],
      dtype=[('a_new', '<i8'), ('b_new', '<f8'), ('c_new', '|S2')])
```

New data types

Likewise the data type for each column can by changed with `dtypes`:

```
>>> Table(arr, dtypes=('f4', 'i4', 'S4'))
<Table rows=2 names=('a','b','c')>
array([(1.0, 2, 'x'), (4.0, 5, 'y')],
      dtype=[('a', '<f4'), ('b', '<i4'), ('c', '|S4')])

>>> Table(arr, names=('a_new', 'b_new', 'c_new'), dtypes=('f4', 'i4', 'S4'))
<Table rows=2 names=('a_new','b_new','c_new')>
array([(1.0, 2, 'x'), (4.0, 5, 'y')],
      dtype=[('a_new', '<f4'), ('b_new', '<i4'), ('c_new', '|S4')])
```

NumPy homogeneous array A normal `numpy` 2-d array (where all elements have the same type) can 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]])
>>> Table(arr)
<Table rows=2 names=('col0','col1','col2')>
array([(1, 2, 3), (4, 5, 6)],
      dtype=[('col0', '<i8'), ('col1', '<i8'), ('col2', '<i8')])
```

Column names and types specified

```
>>> Table(arr, names=('a_new', 'b_new', 'c_new'), dtypes=('f4', 'i4', 'S4'))
<Table rows=2 names=('a_new','b_new','c_new')>
array([(1.0, 2, '3'), (4.0, 5, '6')],
      dtype=[('a_new', '<f4'), ('b_new', '<i4'), ('c_new', '|S4')])
```

Referencing the original data

It is possible to reference the original data for an 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 in 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 an homogeneous `numpy` array input is interpreted as a list of rows:

```
>>> arr = [[1, 2, 3],
...        [4, 5, 6]]
>>> np_arr = np.array(arr)

>>> Table(arr)    # Two columns, three rows
<Table rows=3 names=('col0','col1')>
array([(1, 4), (2, 5), (3, 6)],
      dtype=[('col0', '<i8'), ('col1', '<i8')])

>>> Table(np_arr)  # Three columns, two rows
<Table rows=2 names=('col0','col1','col2')>
array([(1, 2, 3), (4, 5, 6)],
      dtype=[('col0', '<i8'), ('col1', '<i8'), ('col2', '<i8')])
```

This dichotomy is needed to support flexible list input while retaining the natural interpretation of 2-d `numpy` arrays where the first index corresponds to data "rows" and the second index corresponds to data "columns".

If you have a Python list which is structured as a list of data rows, use the following trick to effectively transpose into a list of columns for initializing a `Table` object:

```
>>> arr = [[1, 2.0, 'string'],  # list of rows
           [2, 3.0, 'values']]
>>> col_arr = zip(*arr)  # transpose to a list of columns
>>> col_arr
[(1, 2), (2.0, 3.0), ('string', 'values')]
>>> t = Table(col_arr)
```

Table columns A new table can be created by selecting a subset of columns in an existing table:

```
>>> t = Table(names=('a', 'b', 'c'))
>>> t2 = t['c', 'b', 'a']  # Makes a copy of the data
>>> print t2
<Table rows=0 names=('c','b','a')>
array([],
      dtype=[('c', '<f8'), ('b', '<f8'), ('a', '<f8')])
```

An alternate way to use the columns attribute (explained in the TableColumns section) to initialize a new table. This let's you choose columns by their numerical index or name and supports slicing syntax:

```
>>> Table(t.columns[0:2])
<Table rows=0 names=('a','b')>
array([],
      dtype=[('a', '<f8'), ('b', '<f8')])

>>> Table([t.columns[0], t.columns['c']])
<Table rows=0 names=('a','c')>
array([],
      dtype=[('a', '<f8'), ('c', '<f8')])
```

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, or Table] Data to initialize table.

names
: [list] Specify column names

dtypes
: [list] Specify column data types

meta
: [dict-like] Meta-Data associated with the table

copy
: [boolean] Copy the input data (default=True).

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 dtypes list (optional) must match the length of names and is used to override the existing data types.

numpy ndarray (homogeneous)
: The data ndarray must be at least 2-dimensional, with the first (left-most) index corresponding to row number (table length) and the second index corresponding 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 dtypes 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 dtypes 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 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 `dtypes` list will override the existing or default data types and must match `names` in length.

None
: Initialize a zero-length table. If `names` and optionally `dtypes` 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 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.

dtypes The `dtypes` 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 `dtypes` 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 dict of columns, the `dtypes` argument must be accompanied by a corresponding `names` argument in order to uniquely specify the column ordering.

meta The `meta` argument is simply an object that contains meta-data 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 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 one 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.

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), it is possible to disable copying so that instead a memory reference to the original data is used. 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')])
>>> arr['a']  # column "a" of the input array
array([1, 4])
```

```
>>> t = Table(arr, copy=False)
>>> t['a'][1] = 99
>>> arr['a']  # arr['a'] got changed when we modified t['a']
array([ 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, dtypes=('f4', 'i4', 'S4'))
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
  File "astropy/table/table.py", line 351, in __init__
    raise ValueError('Cannot specify dtypes when copy=False')
ValueError: Cannot specify dtypes when copy=False
```

Another caveat in using referenced data is that you cannot add new row to the table. This generates an error because of conflict between the two references to the same underlying memory. Internally, adding a row may involve moving the data to a new memory location which would corrupt the input data object. `numpy` does not allow this:

```
>>> t.add_row([1, 2, 3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "astropy/table/table.py", line 760, in add_row
    self._data.resize((newlen,), refcheck=False)
ValueError: cannot resize this array: it does not own its data
```

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` is required as the first argument and one can optionally provide these values:

`description`
: [str] Full description of column

`units`
: [str] Physical units

`format`
: [str] Format string for outputting column values

`meta`
: [dict] Meta-data 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 and optionally a ‘‘dtype‘‘ value

Examples:

```
col = Column('a', data=[1, 2, 3])         # shape=(3,)
col = Column('a', data=[[1, 2], [3, 4]])  # shape=(2, 2)
col = Column('a', data=[1, 2, 3], dtype=float)
```

```
col = Column('a', np.array([1, 2, 3]))
col = Column('a', ['hello', 'world'])
```

The dtype argument can be any value which is an acceptable fixed-size data-type initializer for the numpy.dtype() method. See http://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html. Examples include:

- Python non-string type (float, int, bool)
- Numpy non-string type (e.g. np.float32, np.int64, np.bool)
- Numpy.dtype array-protocol type strings (e.g. 'i4', 'f8', 'S15')

If no dtype value is provide then the type is inferred using `np.array(data)`. When data is provided then the shape and length arguments are ignored.

Provide zero or more of ''dtype'', ''shape'', ''length''

Examples:

```
col = Column('a')
col = Column('a', dtype=int, length=10, shape=(3,4))
```

The default dtype is `np.float64` and the default length is zero. 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.

Format string The format string controls the output of column values when a table or column is printed or written to an ASCII table. The format string can be either "old-style" or "new-style":

Old-style

This corresponds to syntax like `"%.4f" % value` as documented in String formatting operations.

`"%.4f"` to print four digits after the decimal in float format, or

`"%6d"` to print an integer in a 6-character wide field.

New-style

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 6-character wide field.

Note that in either case any Python format string that formats exactly one value is valid, so `{:.4f} angstroms` or `Value: %12.2f` would both work.

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 column by index or name

```
>>> t.columns[1]  # Choose columns by index
<Column name='b' units=None format=None description=None>
array([], dtype=float64)

>>> t.columns['b']  # Choose column by name
<Column name='b' units=None format=None description=None>
array([], dtype=float64)
```

Accessing a table

Accessing the table properties and data is straightforward and is generally consistent with the basic interface for numpy structured arrays.

Quick overview

For the impatient, the code below shows the basics of accessing table data. Where relevant there is a comment about what sort of object. Except where noted, the table access returns objects that can be modified in order to update table data or properties. In cases where is returned and how the data contained in that object relate to the original table data (i.e. whether it is a copy or reference, see *Copy versus Reference*).

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
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['a', 'c']  # Table with cols 'a', 'c' (copy)
dat = np.array(t)  # Copy table data to numpy structured array object
```

Print table or column

```
print t      # Print formatted version of table to the screen
t.pprint()   # Same as above
t.pprint(show_units=True)  # Show column units
t.pprint(show_name=False)  # Do not show column names
t.pprint(max_lines=-1, max_width=-1)  # Print full table no matter how long / wide it is

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 valuues as a list
```

Details

For all the following examples it is assumed that the table has been created as below:

```
>>> from astropy.table import Table, Column
>>> import numpy as np

>>> arr = np.arange(15).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'].units = 'm sec^-1'
>>> t['a'].description = 'unladen swallow velocity'
>>> print t
  a     b   c
------ --- ---
 0.000   1   2
 3.000   4   5
 6.000   7   8
 9.000  10  11
12.000  13  14
```

Accessing properties The code below shows accessing the table columns as a `TableColumns` object, getting the column names, table meta-data, and number of table rows. The table meta-data is simply 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 one 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' units='m sec^-1' format='%6.3f' description='unladen swallow velocity'>
array([ 0,  3,  6,  9, 12])

>>> t['a'][1]  # Row 1 of column 'a'
3
```

When a table column is printed, either with `print` or via the `str()` built-in function, it is formatted according to the `format` attribute (see *Format string*):

```
>>> print t['a'].description, t['a']
unladen swallow velocity  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 1 of table
 values=(3, 4, 5)
 dtype=[('a', '<i8'), ('b', '<i8'), ('c', '<i8')]>

>>> t[1]['a']  # Column 'a' of row 1
3
```

A `Row` object has the same columns and meta-data 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 which references to the original data within the slice region (See *Copy versus Reference*). The table meta-data and column definitions are copied.

```
>>> t[2:5]  # Table object with rows 2:5 (reference)
<Table rows=3 names=('a','b','c')>
array([(6, 7, 8), (9, 10, 11), (12, 13, 14)],
      dtype=[('a', '<i8'), ('b', '<i8'), ('c', '<i8')])
```

It is possible to select table rows with an array of indexes or by providing specifying multiple column names. This returns a copy of the original table for the selected rows.

```
>>> print t[[1, 3, 4]]  # Table object with rows 1, 3, 4 (copy)
  a     b   c
------ --- ---
 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
------ --- ---
 3.000   4   5
 9.000  10  11
12.000  13  14

>>> print t['a', 'c']  # Table with cols 'a', 'c' (copy)
  a     c
------ ---
 0.000   2
 3.000   5
 6.000   8
 9.000  11
12.000  14
```

Finally, one 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
```

Formatted printing The values in a table or column can be printed or retrieved as a formatted table using one of several methods:

- print statement (Python 2) or print() function (Python 3).
- 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 column format specifications 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. For example:

```
>>> 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 ... col24 col25 col26 col27 col28 col29
---- ---- ---- ---- ---- ---- ---- ... ----- ----- ----- ----- ----- -----
   0    1    2    3    4    5    6 ...    24    25    26    27    28    29
  30   31   32   33   34   35   36 ...    54    55    56    57    58    59
  60   61   62   63   64   65   66 ...    84    85    86    87    88    89
```

```
  90   91   92   93   94   95   96 ...  114  115  116  117  118  119
 120  121  122  123  124  125  126 ...  144  145  146  147  148  149
 150  151  152  153  154  155  156 ...  174  175  176  177  178  179
 180  181  182  183  184  185  186 ...  204  205  206  207  208  209
 210  211  212  213  214  215  216 ...  234  235  236  237  238  239
 240  241  242  243  244  245  246 ...  264  265  266  267  268  269
 ...  ...  ...  ...  ...  ...  ... ...  ...  ...  ...  ...  ...  ...
2760 2761 2762 2763 2764 2765 2766 ... 2784 2785 2786 2787 2788 2789
2790 2791 2792 2793 2794 2795 2796 ... 2814 2815 2816 2817 2818 2819
2820 2821 2822 2823 2824 2825 2826 ... 2844 2845 2846 2847 2848 2849
2850 2851 2852 2853 2854 2855 2856 ... 2874 2875 2876 2877 2878 2879
2880 2881 2882 2883 2884 2885 2886 ... 2904 2905 2906 2907 2908 2909
2910 2911 2912 2913 2914 2915 2916 ... 2934 2935 2936 2937 2938 2939
2940 2941 2942 2943 2944 2945 2946 ... 2964 2965 2966 2967 2968 2969
2970 2971 2972 2973 2974 2975 2976 ... 2994 2995 2996 2997 2998 2999
```

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_units` with meaning 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'].units = 'km**2'
>>> t['col29'].units = 'kg sec m**-2'

>>> t.pprint(max_lines=8, max_width=40)
    col0         col1    ... col29
------------ ----------- ... ------
0.000000e+00    1.000000 ...   29.0
3.000000e+01   31.000000 ...   59.0
6.000000e+01   61.000000 ...   89.0
         ...         ... ...    ...
2.940000e+03 2941.000000 ... 2969.0
2.970000e+03 2971.000000 ... 2999.0

>>> t.pprint(max_lines=8, max_width=40, show_units=True)
    col0     ...    col29
```

```
   km**2     ... kg sec m**-2
------------ ... ------------
0.000000e+00 ...         29.0
3.000000e+01 ...         59.0
         ... ...          ...
2.940000e+03 ...       2969.0
2.970000e+03 ...       2999.0

>>> t.pprint(max_lines=8, max_width=40, show_name=False)
0.000000e+00    1.000000 ...   29.0
3.000000e+01   31.000000 ...   59.0
6.000000e+01   61.000000 ...   89.0
9.000000e+01   91.000000 ...  119.0
         ...         ... ...    ...
2.910000e+03 2911.000000 ... 2939.0
2.940000e+03 2941.000000 ... 2969.0
2.970000e+03 2971.000000 ... 2999.0
```

In order to force printing all values regardless of the output length or width set `max_lines` or `max_width` to `-1`, respectively. For the wide table in this example one sees 6 lines of wrapped output like the following:

```
>>> t.pprint(max_lines=6, max_width=-1)

    col0         col1     col2   col3   col4   col5   col6   col7   col8   col
9  col10  col11  col12  col13  col14  col15  col16  col17  col18  col19  col20
  col21  col22  col23  col24  col25  col26  col27  col28  col29
------------ ------------ ------ ------ ------ ------ ------ ------ ------ ----
-- ------ ------ ------ ------ ------ ------ ------ ------ ------ ------ -----
- ------ ------ ------ ------ ------ ------ ------ ------ ------
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
3.000000e+01   31.000000   32.0   33.0   34.0   35.0   36.0   37.0   38.0   39
.0   40.0   41.0   42.0   43.0   44.0   45.0   46.0   47.0   48.0   49.0   50.
0   51.0   52.0   53.0   54.0   55.0   56.0   57.0   58.0   59.0
         ...         ...    ...    ...    ...    ...    ...    ...    ...    .
..    ...    ...    ...    ...    ...    ...    ...    ...    ...    ...    ..
.    ...    ...    ...    ...    ...    ...    ...    ...    ...
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
```

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
  63.0
   ...
2943.0
2973.0
```

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.

```
>>> lines = t['col3'].pformat(max_lines=8)
>>> lines
['  col3', '------', '   3.0', '  33.0', '  63.0', '   ...', '2943.0', '2973.0']
```

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 3 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.add_column(Column('a', arr))
>>> t['a'].shape
(3, 2, 2)
>>> t.pprint()
a [2,2]
-------
1 .. 20
3 .. 40
5 .. 60
```

In order to see all the data values for a multidimensional column use the column representation. This uses the standard numpy mechanism for printing any array:

```
>>> t['a']
<Column name='a' units=None format=None description=None>
array([[[ 1,  2],
        [10, 20]],

       [[ 3,  4],
        [30, 40]],

       [[ 5,  6],
        [50, 60]]])
```

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 easily modify the structure of the table: one 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.

Make a table

```
>>> from astropy.table import Table, Column
>>> 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
```

Add a column or columns

```
>>> t.add_column(Column('d', np.arange(5)))

# Make a new table with the same number of rows and add columns to original table
>>> t2 = Table(np.arange(25).reshape(5, 5), names=('e', 'f', 'g', 'h', 'i'))
>>> t.add_columns(t2.columns.values())
```

Remove columns

```
>>> t.remove_column('f')
>>> t.remove_columns(['d', 'e'])
>>> del t['g']
>>> del t['h', 'i']
>>> t.keep_columns(['a', 'b'])
```

Rename columns

```
>>> t.rename_column('a', 'a_new')
>>> t['b'].name = 'b_new'
```

Add a row of data

```
>>> t.add_row([-8, -9])
```

Sort by one more more columns

```
>>> t.sort('b_new')
>>> t.sort(['a_new', 'b_new'])
```

Reverse table rows

```
>>> t.reverse()
```

Modify meta-data

```
>>> t.meta['key'] = 'value'
```

Reorder columns

Note: In this example, it is important that `neworder` is a tuple, and not a list, slice, or `ndarray`.

```
>>> t_acb = t['a','c','b']
>>> neworder = ('a','c','b')
>>> t_acb = t[neworder]
```

Caveats

Modifying the table data and properties is fairly straightforward. There are only a few things to keep in mind:

- The data type for a column cannot be changed in place. In order to do this you must make a copy of the table with the column type changed appropriately.
- Adding or removing a column will generate a new copy in memory of all the data. If the table is very large this may be slow.
- 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.

Masking and missing values

The `astropy.table` package provides support for masking and missing values in a table by wrapping the `numpy.ma` masked array package. 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 the table 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.

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 http://numpy.scipy.org/NA-overview.html for a very interesting discussion of different strategies for handling missing data in the context of `numpy`.

Note: Masked tables are only available for `numpy` version 1.5 and later because of issues in the masked array implementation for prior `numpy` versions.

Table creation

A masked table can be created in several ways:

Create a new table object and specify masked=True

```
>>> from astropy.table import Table, Column, MaskedColumn
>>> t = Table([(1, 2), (3, 4)], names=('a', 'b'), masked=True)
>>> t
<Table rows=2 names=('a','b')>
masked_array(data = [(1, 3) (2, 4)],
             mask = [(False, False) (False, False)],
       fill_value = (999999, 999999),
            dtype = [('a', '<i8'), ('b', '<i8')])
```

Notice the table attributes `mask` and `fill_value` that are available for a masked table.

Create a table with one or more columns as a MaskedColumn object

```
>>> a = MaskedColumn('a', [1, 2])
>>> b = Column('b', [3, 4])
>>> t = Table([a, b])
```

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.

Create a table with one or more columns as a numpy MaskedArray

```
>>> from numpy import ma  # masked array package
>>> a = ma.array([1, 2])
>>> b = [3, 4]
>>> t = Table([a, b], names=('a', 'b'))
```

Add a MaskedColumn object to an existing table

```
>>> a = Column('a', [1, 2])
>>> b = MaskedColumn('b', [3, 4], mask=[True, False])
>>> t = Table([a])
>>> t.add_column(b)
INFO: Upgrading Table to masked Table [astropy.table.table]
```

Note the INFO message because the underlying type of the table is modified in this operation.

Add a new row to an existing table and specify a mask argument

```
>>> a = Column('a', [1, 2])
>>> b = Column('b', [3, 4])
>>> t = Table([a, b])
>>> t.add_row([3, 6], mask=[True, False])
INFO: Upgrading Table to masked Table [astropy.table.table]
```

Convert an existing table to a masked table

```
>>> t = Table([[1, 2], ['x', 'y']])  # standard (unmasked) table
>>> t = Table(t, masked=True)  # convert to masked table
```

Table access

Nearly all the of 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.
- For standard tables a Row object provides a view of the underlying table data so that it is possible to modify a table by modifying the row values. In masked tables this is a copy so that modifying the Row object has no effect on the original table data.

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 actual mask for the table as a whole or a single column can be viewed and modified via the mask attribute:

```
>>> t = Table([(1, 2), (3, 4)], names=('a', 'b'), masked=True)
>>> t.mask['a'] = [False, True]  # Modify table mask (structured 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.

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
```

Reading and writing Table objects

Introduction

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 readers/writers) and new file formats and extensions can be registered with the `Table` class (see Creating a custom reader/writer). After importing the `Table` class:

```
>>> from astropy.table import Table
```

the `read()` method should be used as:

```
>>> t = Table.read(filename, format='format')
```

where `'format'` is the format of the file to read in, e.g.:

```
>>> t = Table.read('photometry.dat', format='daophot')
```

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='format')
```

but as for the `read()` method, the format may be automatically identified in some cases.

Any additional arguments specified will depend on the format (see e.g. see Built-in readers/writers)

Built-in readers/writers

ASCII formats The `read()` and `write()` methods can be used to read and write formats supported by `astropy.io.ascii`:

IPAC IPAC tables can be read with `format='ipac'`:

```
>>> t = Table.read('2mass.tbl', format='ipac')
```

Note that there are different conventions for characters occuring below the position of the | symbol in IPAC tables. By default, any character below a | will be ignored (since this is the current standard), but if you need to read files that assume characters below the | symbols belong to the column before or after the |, you can specify `definition='left'` or `definition='right'` respectively when reading the table (the default is `definition='ignore'`). The following examples demonstrate the different conventions:

- `definition='ignore'`:

```
|   ra  |  dec  |
| float | float |
  1.2345  6.7890
```

- `definition='left'`:

```
|   ra  |  dec  |
| float | float |
   1.2345  6.7890
```

- `definition='right'`:

```
|   ra  |  dec  |
| float | float |
1.2345  6.7890
```

Advanced information is available in the `Ipac` class (any arguments apart from the filename and `format` are passed to this class when `format='ipac'`).

CDS/Machine Readable CDS/Machine readable tables can be read with `format='cds'`:

```
>>> t = Table.read('aj285677t3.txt', format='cds')
```

If the table definition is given in a separate `ReadMe` file, this can be specified with:

```
>>> t = Table.read('aj285677t3.txt', format='cds', readme="ReadMe")
```

Advanced information is available in the `Cds` class (any arguments apart from the filename and `format` are passed to this class when `format='cds'`).

DAOPhot DAOPhot tables can be read with `format='daophot'`:

```
>>> t = Table.read('photometry.dat', format='daophot')
```

Advanced information is available in the `Daophot` class (any arguments apart from the filename and `format` are passed to this class when `format='daophot'`).

LaTeX LaTeX tables can be read and written with `format='latex'`. Provided the `.tex‘` extension is used, the format does not need to be explicitly specified:

```
>>> t = Table.read('paper_table.tex')
>>> t.write('new_paper_table.tex')
```

If a different extension is used, the format should be specified:

```
>>> t.write('new_paper_table.inc', format='latex')
```

Advanced information is available in the Latex class (any arguments apart from the filename and format are passed to this class when format='latex').

RDB RDB tables can be read and written with format='rdb' Provided the .rdb extension is used, the format does not need to be explicitly specified:

```
>>> t = Table.read('discovery_data.rdb')
>>> t.write('updated_data.rdb')
```

If a different extension is used, the format should be specified:

```
>>> t.write('updated_data.txt', format='rdb')
```

Advanced information is available in the Rdb class (any arguments apart from the filename and format are passed to this class when format='rdb').

Arbitrary ASCII formats format='ascii' can be used to interface to the bare read() and write() functions from astropy.io.ascii, e.g.:

```
>>> t = Table.read('table.tex', format='ascii')
```

All additional arguments are passed to the astropy.io.ascii read() and write(). For example, in the following case:

```
>>> t = Table.read('photometry.dat', format='ascii', data_start=2, delimiter='|')
```

the data_start and delimiter arguments are passed to the read() function from astropy.io.ascii (and similarly for writing).

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. For example, 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)
```

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)
```

As with other formats, the `overwrite=True` argument is supported for overwriting existing files.

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.

If a VO table file only contains a single table, then it can be read in with:

```
>>> t = Table.read('aj285677t3_votable.xml')
```

If more that one table are 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):
  File "<stdin>", line 1, in <module>
  File "/Volumes/Raptor/Library/Python/2.7/lib/python/site-packages/astropy/table/table.py", line 1559, in read
    table = reader(*args, **kwargs)
  File "/Volumes/Raptor/Library/Python/2.7/lib/python/site-packages/astropy/io/votable/connect.py", line 44, in read_tabl
    raise ValueError("Multiple tables found: table id should be set via the id= argument. The available tables are " + '
ValueError: Multiple tables found: table id should be set via the table_id= argument. The available tables are twomass, s

>>> 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.

Other In future, FITS tables will also be supported via the `Table` class. For now, these can be read and written directly with `astropy.io.fits`.

Creating a custom reader/writer

The following example demonstrates how to create a reader for the Table class. First, we can create a highly simplistic FITS reader which just reads the data as a structured array:

```
from astropy.table import Table

def fits_reader(filename, hdu=1):
    from astropy.io import fits
    data = fits.open(filename)[hdu].data
    return Table(data)
```

and then register it with astropy.table:

```
from astropy.table import io_registry
io_registry.register_reader('fits', fits_reader)
```

Reader functions can take any arguments except `format` (since this is reserved for the `Table.read` method) and should return a `Table` object.

We can then read in a FITS table with:

```
t = Table.read('catalog.fits', format='fits')
```

In practice, it would be nice to have the `read` method automatically identify that this file was a FITS file, so we can construct a function that can recognize FITS files, which we refer to here as an *identifier* function. An identifier function should take three arguments: the first should be a string which indicates whether the identifier is being called from `read` or `write`, and the second and third are the positional and keyword arguments passed to `Table.read` respectively (and are therefore a list and a dictionary). We can write a simplistic 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 is that it return a boolean indicating whether the input matches that expected for the format:

```
def fits_identify(origin, args, kwargs):
    return isinstance(args[0], basestring) and \
           args[0].lower().split('.')[-1] in ['fits', 'fit']
```

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 with `astropy.table`:

```
io_registry.register_identifier('fits', fits_identify)
```

And we can then do:

```
t = Table.read('catalog.fits')
```

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.

Similarly, it is possible to create custom writers. To go with our simplistic FITS reader above, we can write a simplistic FITS writer:

```
def fits_writer(table, filename, clobber=False):
    import numpy as np
    from astropy.io import fits
    fits.writeto(filename, np.array(table), clobber=clobber)
```

We then register the writer:

```
io_registry.register_writer('fits', fits_writer)
```

And we can then write the file out to a FITS file:

```
t.write('catalog_new.fits', format='fits')
```

If we have registered the identifier as above, we can simply do:

```
t.write('catalog_new.fits')
```

1.8.4 Reference/API

astropy.table Module

Classes

Column	Define a data column for use in a Table object.
MaskedColumn	
Row(table, index)	A class to represent one row of a Table object.
Table([data, masked, names, dtypes, meta, copy])	A class to represent tables of heterogeneous data.
TableColumns([cols])	OrderedDict subclass for a set of columns.

Column

class astropy.table.table.Column

Bases: astropy.table.table.BaseColumn, numpy.ndarray

Define a data column for use in a Table object.

Parameters

name : str

Column name and key for reference within Table

data : list, ndarray or None

Column data values

dtype : numpy.dtype compatible value

Data type for column

shape : tuple or ()

Dimensions of a single row element in the column data

length : int or 0

Number of row elements in column data

description : str or None

Full description of column

units : str or None

Physical units

format : str or None

Format string for outputting column values. This can be an "old-style" (`format % value`) or "new-style" (`str.format`) format specification string.

meta : dict-like or None

Meta-data associated with the column

Examples

A Column can be created in two different ways:

•Provide a `data` value and optionally a `dtype` value

Examples:

```
col = Column('name', data=[1, 2, 3])         # shape=(3,)
col = Column('name', data=[[1, 2], [3, 4]])  # shape=(2, 2)
col = Column('name', data=[1, 2, 3], dtype=float)
col = Column('name', np.array([1, 2, 3]))
col = Column('name', ['hello', 'world'])
```

The `dtype` argument can be any value which is an acceptable fixed-size data-type initializer for the numpy.dtype() method. See http://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html. Examples include:

–Python non-string type (float, int, bool)

–Numpy non-string type (e.g. np.float32, np.int64, np.bool)

–Numpy.dtype array-protocol type strings (e.g. 'i4', 'f8', 'S15')

If no `dtype` value is provide then the type is inferred using `np.array(data)`. When `data` is provided then the `shape` and `length` arguments are ignored.

•Provide zero or more of `dtype`, `shape`, `length`

Examples:

```
col = Column('name')
col = Column('name', dtype=int, length=10, shape=(3,4))
```

The default `dtype` is `np.float64` and the default `length` is zero. 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.

Attributes Summary

`data`

Methods Summary

`copy`([data, copy_data])	Return a copy of the current Column instance.

Attributes Documentation

`data`

Methods Documentation

`copy`(*data=None*, *copy_data=True*)
Return a copy of the current Column instance.

MaskedColumn
class `astropy.table.table.MaskedColumn`
Bases: `astropy.table.table.BaseColumn`, `numpy.ma.core.MaskedArray`

Attributes Summary

`fill_value`
`data`

Methods Summary

`copy`([data, copy_data])	Return a copy of the current MaskedColumn instance.
`filled`([fill_value])	Return a copy of self, with masked values filled with a given value.

Attributes Documentation

`fill_value`

`data`

Methods Documentation

`copy`(*data=None*, *copy_data=True*)
Return a copy of the current MaskedColumn instance.

Parameters
data : array; optional

Data to use when creating MaskedColumn copy. If not supplied the column data array is used.

copy_data : boolean; optional

Make a copy of input data instead of using a reference (default=True)

Returns
column : MaskedColumn

A copy of `self`

`filled`(*fill_value=None*)

Return a copy of self, with masked values filled with a given value.

Parameters

fill_value : scalar; optional

The value to use for invalid entries (None by default). If None, the `fill_value` attribute of the array is used instead.

Returns

filled_column : Column

A copy of `self` with masked entries replaced by `fill_value` (be it the function argument or the attribute of `self`).

Row

class `astropy.table.table.Row`(*table*, *index*)

Bases: `object`

A class to represent one row of a Table object.

A Row object is returned when a Table object is indexed with an integer or when iterating over a table:

```
>>> table = Table([(1, 2), (3, 4)], names=('a', 'b'))
>>> row = table[1]
>>> row
<Row 1 of table
 values=(2, 4)
 dtype=[('a', '<i8'), ('b', '<i8')]>
>>> row['a']
2
>>> row[1]
4
```

Attributes Summary

`dtype`
`meta`
`table`
`colnames`
`data`
`index`
`columns`

Attributes Documentation

`dtype`

`meta`

`table`

colnames

data

index

columns

Table

class astropy.table.table.Table(*data=None*, *masked=None*, *names=None*, *dtypes=None*, *meta=None*, *copy=True*)

Bases: object

A class to represent tables of heterogeneous data.

Table provides a class for heterogeneous tabular data, making use of a numpy structured array internally to store the data values. A key enhancement provided by the Table class is the ability to easily modify the structure of the table by adding or removing columns, or adding new rows of data. In addition table and column metadata are fully supported.

Table differs from NDData by the assumption that the input data consists of columns of homogeneous data, where each column has a unique identifier and may contain additional metadata such as the data units, format, and description.

Parameters

data : numpy ndarray, dict, list, or Table, optional

Data to initialize table.

mask : numpy ndarray, dict, list, optional

The mask to initialize the table

names : list, optional

Specify column names

dtypes : list, optional

Specify column data types

meta : dict, optional

Metadata associated with the table

copy : boolean, optional

Copy the input data (default=True).

Attributes Summary

masked
colnames
dtype
ColumnClass
mask

Methods Summary

create_mask()	
rename_column(name, new_name)	Rename a column.
pprint([max_lines, max_width, show_name, ...])	Print a formatted string representation of the table.
index_column(name)	Return the positional index of column name.
next()	Python 3 iterator
write(*args, **kwargs)	Write a table
field(item)	Return column[item] for recarray compatibility.
add_column(col[, index])	Add a new Column object col to the table.
filled([fill_value])	Return a copy of self, with masked values filled.
more([max_lines, max_width, show_name, ...])	Interactively browse table with a paging interface.
sort(keys)	Sort the table according to one or more keys.
keys()	
remove_columns(names)	Remove several columns from the table
reverse()	Reverse the row order of table rows.
read(*args, **kwargs)	Read a table
add_columns(cols[, indexes])	Add a list of new Column objects cols to the table.
keep_columns(names)	Keep only the columns specified (remove the others).
remove_column(name)	Remove a column from the table.
pformat([max_lines, max_width, show_name, ...])	Return a list of lines for the formatted string representation of the table.
add_row([vals, mask])	Add a new row to the end of the table.

Attributes Documentation

masked

colnames

dtype

ColumnClass

mask

Methods Documentation

create_mask()

rename_column(*name*, *new_name*)

Rename a column.

This can also be done directly with by setting the name attribute for a column:

```
table[name].name = new_name
```

Parameters

name : str

The current name of the column.

new_name : str

The new name for the column

pprint(*max_lines=None*, *max_width=None*, *show_name=True*, *show_units=False*)

Print a formatted string representation of the table.

If no value of max_lines is supplied then the height of the screen terminal is used to set max_lines. If the terminal height cannot be determined then the default is taken from the configuration item astropy.table.pprint.MAX_LINES. If a negative value of max_lines is supplied then there is no line limit applied.

The same applies for max_width except the configuration item is astropy.table.pprint.MAX_WIDTH.

Parameters

max_lines : int

Maximum number of lines in table output

max_width : int or None

Maximum character width of output

show_name : bool

Include a header row for column names (default=True)

show_units : bool

Include a header row for units (default=False)

index_column(*name*)

Return the positional index of column name.

Parameters

name : str

column name

Returns

index : int

Positional index of column name.

next()

Python 3 iterator

write(**args*, ***kwargs*)

Write a table

The arguments passed to this method depend on the format

field(*item*)

Return column[item] for recarray compatibility.

add_column(*col*, *index=None*)

Add a new Column object col to the table. If index is supplied then insert column before index position in the list of columns, otherwise append column to the end of the list.

Parameters

col : Column

Column object to add.

index : int or None

Insert column before this position or at end (default)

`filled(`*fill_value=None*`)`

Return a copy of self, with masked values filled.

If input `fill_value` supplied then that value is used for all masked entries in the table. Otherwise the individual `fill_value` defined for each table column is used.

Returns

filled_table : Table

New table with masked values filled

`more(`*max_lines=None*, *max_width=None*, *show_name=True*, *show_units=False*`)`

Interactively browse table with a paging interface.

Supported keys:

```
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
```

Parameters

max_lines : int

Maximum number of lines in table output

max_width : int or None

Maximum character width of output

show_name : bool

Include a header row for column names (default=True)

show_units : bool

Include a header row for units (default=False)

`sort(`*keys*`)`

Sort the table according to one or more keys. This operates on the existing table and does not return a new table.

Parameters

keys : str or list of str

The key(s) to order the table by

`keys()`

`remove_columns(`*names*`)`

Remove several columns from the table

Parameters

names : list

A list containing the names of the columns to remove

`reverse()`
Reverse the row order of table rows. The table is reversed in place and there are no function arguments.

classmethod `read(*args, **kwargs)`
Read a table

The arguments passed to this method depend on the format

`add_columns(cols, indexes=None)`
Add a list of new Column objects `cols` to the table. If a corresponding list of `indexes` is supplied then insert column before each `index` position in the *original* list of columns, otherwise append columns to the end of the list.

Parameters
cols : list of Columns

Column objects to add.

indexes : list of ints or None

Insert column before this position or at end (default)

`keep_columns(names)`
Keep only the columns specified (remove the others).

Parameters
names : list

A list containing the names of the columns to keep. All other columns will be removed.

`remove_column(name)`
Remove a column from the table.

This can also be done with:

```
del table[name]
```

Parameters
name : str

Name of column to remove

`pformat(max_lines=None, max_width=None, show_name=True, show_units=False, html=False)`
Return a list of lines for the formatted string representation of the table.

If no value of `max_lines` is supplied then the height of the screen terminal is used to set `max_lines`. If the terminal height cannot be determined then the default is taken from the configuration item `astropy.table.pprint.MAX_LINES`. If a negative value of `max_lines` is supplied then there is no line limit applied.

The same applies for max_width except the configuration item is `astropy.table.pprint.MAX_WIDTH`.

Parameters
max_lines : int or None

Maximum number of rows to output

max_width : int or None

Maximum character width of output

show_name : bool

Include a header row for column names (default=True)

show_units : bool

Include a header row for units (default=False)

html : bool

Format the output as an HTML table (default=False)

Returns

lines : list

Formatted table as a list of strings

`add_row`(*vals=None*, *mask=None*)

Add a new row to the end of the table.

The `vals` argument can be:

sequence (e.g. tuple or list)

Column values in the same order as table columns.

mapping (e.g. dict)

Keys corresponding to column names. Missing values will be filled with np.zeros for the column dtype.

None

All values filled with np.zeros for the column dtype.

This method requires that the Table object "owns" the underlying array data. In particular one cannot add a row to a Table that was initialized with copy=False from an existing array.

The `mask` attribute should give (if desired) the mask for the values. The type of the mask should match that of the values, i.e. if `vals` is an iterable, then `mask` should also be an iterable with the same length, and if `vals` is a mapping, then `mask` should be a dictionary.

Parameters

vals : tuple, list, dict or None

Use the specified values in the new row

TableColumns

class `astropy.table.table.TableColumns`(*cols={}*)

Bases: `collections.OrderedDict`

OrderedDict subclass for a set of columns.

This class enhances item access to provide convenient access to columns by name or index, including slice access. It also handles renaming of columns.

The initialization argument `cols` can be any structure that is valid for initializing a Python dict. This includes a dict, list of (key, val) tuple pairs, list of [key, val] lists, etc.

Parameters

cols : dict, list, tuple; optional

Column objects as data structure that can init dict (see above)

Methods Summary

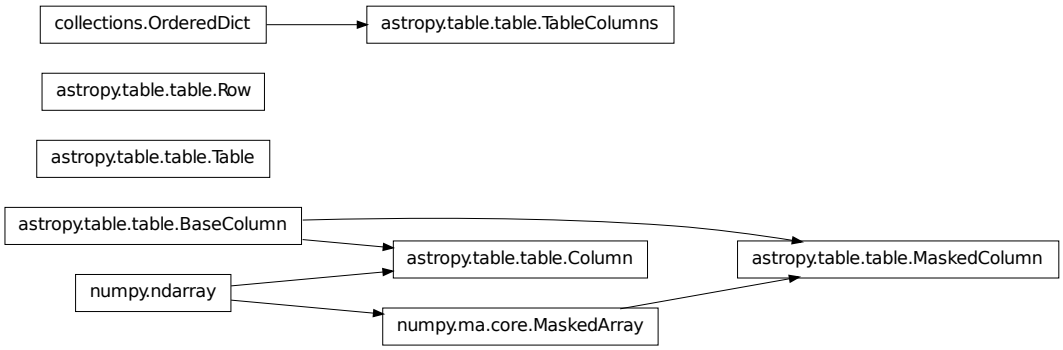
keys()
values()

Methods Documentation

keys()

values()

Class Inheritance Diagram



astropy.table.io_registry Module

Functions

register_reader(table_format, function[, force])	Register a table reader function.
register_writer(table_format, function[, force])	Register a table writer function.
register_identifier(table_format, identifier)	Associate an identifier function with a specific table type.
identify_format(origin, args, kwargs)	
get_reader(table_format)	
get_writer(table_format)	

register_reader
astropy.table.io_registry.register_reader(*table_format*, *function*, *force=False*)
Register a table reader function.

Parameters
table_format : string

The table type identifier. This is the string that will be used to specify the table type when reading.

function : function

The function to read in a table.

force : bool

Whether to override any existing function if already present.

register_writer
`astropy.table.io_registry.register_writer`(*table_format*, *function*, *force=False*)
Register a table writer function.

Parameters

table_format : string

The table type identifier. This is the string that will be used to specify the table type when writing.

function : function

The function to write in a table.

force : bool

Whether to override any existing function if already present.

register_identifier
`astropy.table.io_registry.register_identifier`(*table_format*, *identifier*, *force=False*)
Associate an identifier function with a specific table type.

Parameters

table_format : str

The table type identifier. This is the string that is used to specify the table type when reading/writing.

identifier : function

A function that checks the argument specified to `Table.read` or `Table.write` to determine whether the input can be interpreted as a table of type `table_format`. This function should take two arguments, which will be set to the list of arguments and a dictionary of keyword arguments passed to `Table.read` or `Table.write`. The function should return True if the input can be identified as being of format `table_format`, and False otherwise.

force : bool

Whether to override any existing function if already present.

Examples

To set the identifier based on extensions, for formats that take a filename as a first argument, you can do for example:

```
>>> register_identifier('ipac', lambda args, kwargs: isinstance(args[0], basestring) and args[0].endswith('.tbl'))
```

identify_format
`astropy.table.io_registry.identify_format`(*origin*, *args*, *kwargs*)

get_reader
`astropy.table.io_registry.get_reader(`*table_format*`)`

get_writer
`astropy.table.io_registry.get_writer(`*table_format*`)`

1.9 Cosmological Calculations (`astropy.cosmology`)

1.9.1 Introduction

The `astropy.cosmology` subpackage 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.

1.9.2 Getting Started

There are many functions available to calculate cosmological quantities. They generally take a redshift as input. For example, the two cases below give you the value of the hubble constant at z=0 (i.e., `H0`), and the number of transverse proper kpc corresponding to an arcminute at z=3:

```
>>> from astropy import cosmology
>>> cosmology.H(0)
70.4
>>> cosmology.kpc_proper_per_arcmin(3)
472.8071851564037
```

All the functions available are listed in the Reference/API section. These will use the "current" cosmology to calculate the values (see The Current Cosmology section below for more details). If you haven't set this explicitly, they will use the 7-year WMAP cosmological parameters and print a warning message.

There are also several standard cosmologies already defined. These are objects with methods and attributes that calculate cosmological values. For example, the comoving distance in Mpc to redshift 4 using the 5-year WMAP parameters:

```
>>> from astropy.cosmology import WMAP5
>>> WMAP5.comoving_distance(4)
7329.328120760829
```

A full list of the pre-defined cosmologies is given by `cosmology.parameters.available`.

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.

1.9.3 Using `cosmology`

Most of the functionality is enabled by the `FLRW` object. This represents a homogenous and isotropic cosmology (a 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 can't work with this class directly, as you must specify a dark energy model by using one of its subclasses instead, such as `FlatLambdaCDM`.

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
LambdaCDM(H0=70, Om0=0.3, Ode0=0.7)
```

A number of additional dark energy models are provided (described below). Note that photons and neutrinos are included in these models, so Om0 + Ode0 is not quite one.

The pre-defined cosmologies described in the Getting Started section are instances of FlatLambdaCDM, and have the same methods. So we can find the luminosity distance in Mpc to redshift 4 by:

```
>>> cosmo.luminosity_distance(4)
35842.35374316948
```

or the age of the universe at z = 0 in Gyr:

```
>>> cosmo.age(0)
13.461701807287566
```

They also accept arrays of redshifts:

```
>>> cosmo.age([0.5, 1, 1.5])
array([ 8.42128059,  5.74698062,  4.1964541 ])
```

See the FLRW and FlatLambdaCDM object docstring for all the methods and attributes available. In addition to flat Universes, non-flat varieties are supported such as LambdaCDM. There are also a variety of standard cosmologies with the parameters already defined:

```
>>> from astropy.cosmology import WMAP7   # WMAP 7-year cosmology
>>> WMAP7.critical_density(0)       # critical density at z = 0 in g/cm^3
9.31000313202047e-30


>>> from astropy.cosmology import WMAP5   # WMAP 5-year
>>> WMAP5.H(3)                     # Hubble parameter at z = 3 in km/s/Mpc
301.71804314602889
```

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     ,  0.74898525,  0.9090524 ]),
 array([ 0.72791572,  0.25055062,  0.09010261]))
```

Note that these don't quite add up to one even though WMAP7 assumes a flat Universe because photons and neutrinos are included.

In addition to the LambdaCDM object, there are convenience functions that calculate some of these quantities without needing to explicitly give a cosmology - but there are more methods available if you work directly with the cosmology object.

```
>>> from astropy import cosmology
>>> cosmology.kpc_proper_per_arcmin(3)
472.8071851564037
>>> cosmology.arcsec_per_kpc_proper(3)
0.12690162477152736
```

These functions will perform calculations using the "current" cosmology. This is a specific cosmology that is currently active in astropy and it's described further in the following section. They can also be explicitly given a cosmology using the cosmo keyword argument. A full list of convenience functions is included below, in the Reference/API section.

1.9.4 The Current Cosmology

Sometimes it's useful for Astropy functions to assume a default cosmology so that the desired cosmology doesn't have to be specified every time the function is called – the convenience functions described in the previous section are one example. For these cases it's possible to specify a "current" cosmology.

You can set the current cosmology to a pre-defined 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_current function to set a cosmology for the current Python session.

If you haven't set a current cosmology using one of the methods described above, then the cosmology module will use the 7-year WMAP parameters and print a warning message letting you know this. For example, if you call a convenience function without setting the current cosmology or using the cosmo= keyword you see the following message:

```
>>> from astropy import cosmology
>>> cosmology.lookback_time(1)          # lookback time in Gyr at z=1
WARNING: No default cosmology has been specified, using 7-year WMAP.
[astropy.cosmology.core]
7.787767002228743
```

Note: In general it's better to use an explicit cosmology (for example WMAP7.H(0) instead of cosmology.H(0)). The motivation for this is that when you go back to use the code at a later date or share your scripts with someone else, the default cosmology may have changed. Use of the convenience functions should generally be reserved for interactive work or cases where the flexibility of quickly changing between different cosmologies is for some reason useful. Alternatively, putting (for example) cosmology.set_current(WMAP7) at the top of your code will ensure that the right cosmology is always used.

Using cosmology inside Astropy

If you are writing code for the astropy core or an affiliated package, it is strongly recommended that you use the the current cosmology through the get_current function. It is also recommended that you provide an override option something like the following:

```
def myfunc(..., cosmo=None):
    from astropy.cosmology import get_current

    if cosmo is None:
        cosmo = get_current()

    ... your code here ...
```

This ensures that all code consistently uses the current cosmology unless explicitly overridden.

1.9.5 Specifying a dark energy model

In addition to the standard FlatLambdaCDM model described above, a number of additional dark energy models are provided. FlatLambdaCDM and FlatLambdaCDM assume that dark energy is a cosmological constant, and should be the most commonly used case. wCDM assumes 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 sub-classing FLRW. See the provided subclasses for examples.

1.9.6 Relativistic Species

The cosmology classes include the contribution to the energy density from both photons and massless neutrinos. The two parameters controlling the properties of these species are Tcmb0 (the temperature of the CMB at z=0) and Neff, the effective number of neutrino species. Both have standard default values (2.725 and 3.04, respectively; the reason that Neff is not 3 has to do with a small bump in the neutrino energy spectrum due to electron-positron annihilation).

```
>>> from astropy.cosmology import WMAP7   # WMAP 7-year cosmology
>>> z = [0,1.0,2.0]
>>> WMAP7.Ogamma(z), WMAP7.Onu(z)
(array([  4.98569503e-05,   2.74574414e-04]),
 array([  3.44204408e-05,   1.89561782e-04]),
 array([  8.42773911e-05,   4.64136197e-04]))
```

If you want to exclude photons and neutrinos from your calculations, simply set the CMB Temperature to 0:

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> cos = FlatLambdaCDM(70.4, 0.272, Tcmb0 = 0.0)
>>> cos.Ogamma0, cos.Onu0
(0.0, 0.0)
```

Neutrinos can be removed (while leaving photons) by setting Neff=0:

```
>>> from astropy.cosmology import FlatLambdaCDM
>>> cos = FlatLambdaCDM(70.4, 0.272, Neff=0)
>>> cos.Ogamma([0,1,2]),cos.Onu([0,1,2])
(array([  4.98569503e-05,   2.74623219e-04,   5.00051845e-04]),
 array([ 0.,  0.,  0.]))
```

While these examples used FlatLambdaCDM, the above examples also apply for all of the other cosmology classes.

1.9.7 See Also

- Hogg, "Distance measures in cosmology", http://arxiv.org/abs/astroph/9905116
- Linder, "Exploring the Expansion History of the Universe", http://arxiv.org/abs/astro-ph/0208512
- NASA's Legacy Archive for Microwave Background Data Analysis, http://lambda.gsfc.nasa.gov/

1.9.8 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 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`. Note that the energy density due to radiation is assumed to be negligible, which is valid for redshifts less than about 10. If you find any bugs, please let us know by opening an issue at the github repository!

1.9.9 Reference/API

astropy.cosmology Module

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

H(z[, cosmo])	Hubble parameter (km/s/Mpc) at redshift z.
angular_diameter_distance(z[, cosmo])	Angular diameter distance in Mpc at a given redshift.
arcsec_per_kpc_comoving(z[, cosmo])	Angular separation in arcsec corresponding to a comoving kpc at redshift z.
arcsec_per_kpc_proper(z[, cosmo])	Angular separation in arcsec corresponding to a proper kpc at redshift z.
comoving_distance(z[, cosmo])	Comoving distance in Mpc at redshift z.
critical_density(z[, cosmo])	Critical density in grams per cubic cm at redshift z.
distmod(z[, cosmo])	Distance modulus at redshift z.
get_current()	Get the current cosmology.
kpc_comoving_per_arcmin(z[, cosmo])	Separation in transverse comoving kpc corresponding to an arcminute at redshift z.
kpc_proper_per_arcmin(z[, cosmo])	Separation in transverse proper kpc corresponding to an arcminute at redshift z.
lookback_time(z[, cosmo])	Lookback time in Gyr to redshift z.
luminosity_distance(z[, cosmo])	Luminosity distance in Mpc at redshift z.
scale_factor(z[, cosmo])	Scale factor at redshift z.
set_current(cosmo)	Set the current cosmology.

H

`astropy.cosmology.funcs.H(`*z*, *cosmo=None*`)`

Hubble parameter (km/s/Mpc) at redshift z.

Parameters

z : array_like

Input redshifts.

Returns

H : ndarray, or float if input scalar

Hubble parameter at each input redshift.

angular_diameter_distance

`astropy.cosmology.funcs.angular_diameter_distance(`*z*, *cosmo=None*`)`

Angular diameter distance in Mpc at a given redshift.

This gives the proper (sometimes called 'physical') transverse distance corresponding to an angle of 1 radian for an object at redshift z.

Parameters

z : array_like

Input redshifts.

Returns

angdist : ndarray, or float if input scalar

Angular diameter distance at each input redshift.

arcsec_per_kpc_comoving

`astropy.cosmology.funcs.arcsec_per_kpc_comoving(`*z*, *cosmo=None*`)`

Angular separation in arcsec corresponding to a comoving kpc at redshift z.

Parameters

z : array_like

Input redshifts.

Returns

theta : ndarray, or float if input scalar

The angular separation in arcsec corresponding to a comoving kpc at each input redshift.

arcsec_per_kpc_proper

`astropy.cosmology.funcs.arcsec_per_kpc_proper(`*z*, *cosmo=None*`)`

Angular separation in arcsec corresponding to a proper kpc at redshift z.

Parameters

z : array_like

Input redshifts.

Returns

theta : ndarray, or float if input scalar

The angular separation in arcsec corresponding to a proper kpc at each input redshift.

comoving_distance

`astropy.cosmology.funcs.comoving_distance(`*z*, *cosmo=None*`)`

Comoving distance in Mpc at redshift z.

The comoving distance along the line-of-sight between two objects remains constant with time for objects in the Hubble flow.

Parameters

z : array_like

Input redshifts.

Returns

codist : ndarray, or float if input scalar

Comoving distance at each input redshift.

critical_density
astropy.cosmology.funcs.critical_density(*z*, *cosmo=None*)
Critical density in grams per cubic cm at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
critdens : ndarray, or float if input scalar

Critical density at each input redshift.

distmod
astropy.cosmology.funcs.distmod(*z*, *cosmo=None*)
Distance modulus at redshift z.

The distance modulus is defined as the (apparent magnitude - absolute magnitude) for an object at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
distmod : ndarray, or float if input scalar

Distance modulus at each input redshift.

get_current
astropy.cosmology.core.get_current()
Get the current cosmology.

If no current has been set, the WMAP7 comology is returned and a warning is given.

Returns
cosmo : Cosmology instance

See Also:

set_current
sets the current cosmology

kpc_comoving_per_arcmin
astropy.cosmology.funcs.kpc_comoving_per_arcmin(*z*, *cosmo=None*)
Separation in transverse comoving kpc corresponding to an arcminute at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
d : ndarray, or float if input scalar

The distance in comoving kpc corresponding to an arcmin at each input redshift.

kpc_proper_per_arcmin
`astropy.cosmology.funcs.kpc_proper_per_arcmin(`*z*, *cosmo=None*`)`
Separation in transverse proper kpc corresponding to an arcminute at redshift `z`.

Parameters
z : array_like

Input redshifts.

Returns
d : ndarray, or float if input scalar

The distance in proper kpc corresponding to an arcmin at each input redshift.

lookback_time
`astropy.cosmology.funcs.lookback_time(`*z*, *cosmo=None*`)`
Lookback time in Gyr to redshift `z`.

The lookback time is the difference between the age of the Universe now and the age at redshift `z`.

Parameters
z : array_like

Input redshifts.

Returns
t : ndarray, or float if input scalar

Lookback time at each input redshift.

luminosity_distance
`astropy.cosmology.funcs.luminosity_distance(`*z*, *cosmo=None*`)`
Luminosity distance in Mpc at redshift `z`.

This is the distance to use when converting between the bolometric flux from an object at redshift `z` and its bolometric luminosity.

Parameters
z : array_like

Input redshifts.

Returns
lumdist : ndarray, or float if input scalar

Angular diameter distance at each input redshift.

scale_factor
`astropy.cosmology.funcs.scale_factor(`*z*, *cosmo=None*`)`
Scale factor at redshift `z`.

The scale factor is defined as `a = 1 / (1 + z)`.

Parameters
z : array_like

Input redshifts.

Returns
distmod : ndarray, or float if input scalar

Scale factor at each input redshift.

set_current
`astropy.cosmology.core.set_current(`*cosmo*`)`
Set the current cosmology.

Call this with an empty string (‘’) to get a list of the strings that map to available pre-defined cosmologies.

> **Warning:** `set_current` is the only way to change the current cosmology at runtime! The current cosmology can also be read from an option in the astropy configuration file when astropy.cosmology is first imported. However, any subsequent changes to the cosmology configuration option using `ConfigurationItem.set` at run-time will not update the current cosmology.

Parameters
cosmo : str or `Cosmology` instance

The cosmology to use.

See Also:

`get_current`
returns the currently-set cosmology

Classes

`FLRW`(H0, Om0, Ode0[, Tcmb0, Neff, name])	A class describing an isotropic and homogeneous (Friedmann-Lemaitre-Ro
`FlatLambdaCDM`(H0, Om0[, Tcmb0, Neff, name])	FLRW cosmology with a cosmological constant and no curvature.
`Flatw0waCDM`(H0, Om0[, w0, wa, Tcmb0, Neff, name])	FLRW cosmology with a CPL dark energy equation of state and no curvatu
`FlatwCDM`(H0, Om0[, w0, Tcmb0, Neff, name])	FLRW cosmology with a constant dark energy equation of state and no spat
`LambdaCDM`(H0, Om0, Ode0[, Tcmb0, Neff, name])	FLRW cosmology with a cosmological constant and curvature.
`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 curvatu
`wCDM`(H0, Om0, Ode0[, w0, Tcmb0, Neff, name])	FLRW cosmology with a constant dark energy equation of state and curvatu
`wpwaCDM`(H0, Om0, Ode0[, wp, wa, zp, Tcmb0, ...])	FLRW cosmology with a CPL dark energy equation of state, a pivot redshif

FLRW
class `astropy.cosmology.core.FLRW(`*H0*, *Om0*, *Ode0*, *Tcmb0=2.725*, *Neff=3.04*, *name='FLRW'*`)`
Bases: `astropy.cosmology.core.Cosmology`

A class describing an isotropic and homogeneous (Friedmann-Lemaitre-Robertson-Walker) cosmology.

This is an abstract base class – you can’t instantiate examples of this class, but must work with one of its subclasses such as `LambdaCDM` or `wCDM`.

Notes

Class instances are static – you can’t change the values of the parameters. That is, all of the attributes above are read only.

The neutrino treatment assumes all neutrino species are massless.

Attributes Summary

Neff	Number of effective neutrino species
critical_density0	Critical density in [g cm^-3] at z=0
Ogamma0	Omega gamma; the density/critical density of photons at z=0
hubble_distance	Hubble distance in [Mpc]
Tcmb0	Temperature of the CMB in Kelvin at z=0
hubble_time	Hubble time in [Gyr]
Onu0	Omega nu; the density/critical density of neutrinos at z=0
Ode0	Omega dark energy; dark energy density/critical density at z=0
Ok0	Omega curvature; the effective curvature density/critical density
h	Dimensionless Hubble constant: h = H_0 / 100 [km/sec/Mpc]
Om0	Omega matter; matter density/critical density at z=0
H0	Return the Hubble constant in [km/sec/Mpc] at z=0

Methods Summary

angular_diameter_distance(z)	Angular diameter distance in Mpc at a given redshift.
Ok(z)	Return the equivalent density parameter for curvature at redshift z.
Ogamma(z)	Return the density parameter for photons at redshift z.
comoving_transverse_distance(z)	Comoving transverse distance in Mpc at a given redshift.
critical_density(z)	Critical density in grams per cubic cm at redshift z.
lookback_time(z)	Lookback time in Gyr to redshift z.
absorption_distance(z)	Absorption distance at redshift z.
inv_efunc(z)	Inverse of efunc.
Tcmb(z)	Return the CMB temperature at redshift z.
de_density_scale(z)	Evaluates the redshift dependence of the dark energy density.
Om(z)	Return the density parameter for non-relativistic matter at redshift z.
scale_factor(z)	Scale factor at redshift z.
H(z)	Hubble parameter (km/s/Mpc) at redshift z.
efunc(z)	Function used to calculate H(z), the Hubble parameter.
distmod(z)	Distance modulus at redshift z.
comoving_distance(z)	Comoving line-of-sight distance in Mpc at a given redshift.
age(z)	Age of the universe in Gyr at redshift z.
Ode(z)	Return the density parameter for dark energy at redshift z.
luminosity_distance(z)	Luminosity distance in Mpc at redshift z.
angular_diameter_distance_z1z2(z1, z2)	Angular diameter distance between objects at 2 redshifts.
w(z)	The dark energy equation of state.
comoving_volume(z)	Comoving volume in cubic Mpc at redshift z.
Onu(z)	Return the density parameter for massless neutrinos at redshift z.

Attributes Documentation

Neff
: Number of effective neutrino species

critical_density0
: Critical density in [g cm^-3] at z=0

Ogamma0
: Omega gamma; the density/critical density of photons at z=0

hubble_distance

Hubble distance in [Mpc]

`Tcmb0`
Temperature of the CMB in Kelvin at z=0

`hubble_time`
Hubble time in [Gyr]

`Onu0`
Omega nu; the density/critical density of neutrinos at z=0

`Ode0`
Omega dark energy; dark energy density/critical density at z=0

`Ok0`
Omega curvature; the effective curvature density/critical density at z=0

`h`
Dimensionless Hubble constant: h = H_0 / 100 [km/sec/Mpc]

`Om0`
Omega matter; matter density/critical density at z=0

`H0`
Return the Hubble constant in [km/sec/Mpc] at z=0

Methods Documentation

`angular_diameter_distance(z)`
Angular diameter distance in Mpc at a given redshift.

This gives the proper (sometimes called 'physical') transverse distance corresponding to an angle of 1 radian for an object at redshift z.

Weinberg, 1972, pp 421-424; Weedman, 1986, pp 65-67; Peebles, 1993, pp 325-327.

Parameters
z : array_like

Input redshifts.

Returns
d : ndarray, or float if input scalar

Angular diameter distance in Mpc at each input redshift.

`Ok(z)`
Return the equivalent density parameter for curvature at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
Ok : ndarray, or float if input scalar

The equivalent density parameter for curvature at each redshift.

`Ogamma(z)`
Return the density parameter for photons at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
Ogamma : ndarray, or float if input scalar

The energy density of photons relative to the critical density at each redshift.

`comoving_transverse_distance(z)`
Comoving transverse distance in Mpc at a given redshift.

This value is the transverse comoving distance at redshift z corresponding to an angular separation of 1 radian. This is the same as the comoving distance if omega_k is zero (as in the current concordance lambda CDM model).

Parameters
z : array_like

Input redshifts.

Returns
d : ndarray, or float if input scalar

Comoving transverse distance in Mpc at each input redshift.

Notes

This quantity also called the 'proper motion distance' in some texts.

`critical_density(z)`
Critical density in grams per cubic cm at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
rho : ndarray, or float if input scalar

Critical density in g/cm^3 at each input redshift.

`lookback_time(z)`
Lookback time in Gyr to redshift z.

The lookback time is the difference between the age of the Universe now and the age at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
t : ndarray, or float if input scalar

Lookback time in Gyr to each input redshift.

`absorption_distance(z)`
Absorption distance at redshift z.

This is used to calculate the number of objects with some cross section of absorption and number density intersecting a sightline per unit redshift path.

Parameters
z : array_like

Input redshifts.

Returns
d : ndarray, or float if input scalar

Absorption distance (dimensionless) at each input redshift.

References

Hogg 1999 Section 11. (astro-ph/9905116) Bahcall, John N. and Peebles, P.J.E. 1969, ApJ, 156L, 7B

`inv_efunc(`z`)`
Inverse of efunc.

Parameters
z : array_like

Input redshifts.

Returns
E : ndarray, or float if input scalar

The redshift scaling of the inverse Hubble constant.

`Tcmb(`z`)`
Return the CMB temperature at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
Tcmb : ndarray, or float if z is scalar

The temperature of the CMB in K.

`de_density_scale(`z`)`
Evaluates the redshift dependence of the dark energy density.

Parameters
z : array_like

Input redshifts.

Returns
I : ndarray, or float if input scalar

The scaling of the energy density of dark energy with redshift.

Notes

The scaling factor, I, is defined by $\rho(z) = \rho_0 I$, and is given by

$$I = \exp\left(3\int_a^1 \frac{da'}{a'}\left[1 + w\left(a'\right)\right]\right)$$

It will generally helpful for subclasses to overload this method if the integral can be done analytically for the particular dark energy equation of state that they implement.

`Om(z)`
Return the density parameter for non-relativistic matter at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
Om : ndarray, or float if input scalar

The density of non-relativistic matter relative to the critical density at each redshift.

`scale_factor(z)`
Scale factor at redshift z.

The scale factor is defined as $a = 1/(1+z)$.

Parameters
z : array_like

Input redshifts.

Returns
a : ndarray, or float if input scalar

Scale factor at each input redshift.

`H(z)`
Hubble parameter (km/s/Mpc) at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
H : ndarray, or float if input scalar

Hubble parameter in km/s/Mpc at each input redshift.

`efunc(z)`
Function used to calculate H(z), the Hubble parameter.

Parameters
z : array_like

Input redshifts.

Returns
E : ndarray, or float if input scalar

The redshift scaling of the Hubble constant.

Notes

The return value, E, is defined such that $H(z) = H_0 E$.

It is not necessary to override this method, but if de_density_scale takes a particularly simple form, it may be advantageous to.

distmod(*z*)
Distance modulus at redshift z.

The distance modulus is defined as the (apparent magnitude - absolute magnitude) for an object at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
distmod : ndarray, or float if input scalar

Distance modulus at each input redshift.

comoving_distance(*z*)
Comoving line-of-sight distance in Mpc at a given redshift.

The comoving distance along the line-of-sight between two objects remains constant with time for objects in the Hubble flow.

Parameters
z : array_like

Input redshifts.

Returns
d : ndarray, or float if input scalar

Comoving distance in Mpc to each input redshift.

age(*z*)
Age of the universe in Gyr at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
t : ndarray, or float if input scalar

The age of the universe in Gyr at each input redshift.

Ode(*z*)
Return the density parameter for dark energy at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
Ode : ndarray, or float if input scalar

The density of non-relativistic matter relative to the critical density at each redshift.

luminosity_distance(*z*)
Luminosity distance in Mpc at redshift z.

This is the distance to use when converting between the bolometric flux from an object at redshift z and its bolometric luminosity.

Parameters
z : array_like

Input redshifts.

Returns
d : ndarray, or float if input scalar

Luminosity distance in Mpc at each input redshift.

References

Weinberg, 1972, pp 420-424; Weedman, 1986, pp 60-62.

`angular_diameter_distance_z1z2(`*z1*, *z2*`)`
Angular diameter distance between objects at 2 redshifts. Useful for gravitational lensing.

Parameters
z1, z2 : array_like, shape (N,)

Input redshifts. z2 must be large than z1.

Returns
d : ndarray, shape (N,) or float if input scalar

The angular diameter distance between each input redshift pair.

Raises
CosmologyError :

If omega_k is < 0.

Notes

This method only works for flat or open curvature (omega_k >= 0).

`w(`*z*`)`
The dark energy equation of state.

Parameters
z : array_like

Input redshifts.

Returns
w : ndarray, or float if input scalar

The dark energy equation of state

Notes

The dark energy equation of state is defined as $w(z) = P(z)/\rho(z)$, where $P(z)$ is the pressure at redshift z and $\rho(z)$ is the density at redshift z, both in units where c=1.

This must be overridden by subclasses.

`comoving_volume(`*z*`)`
Comoving volume in cubic Mpc at redshift z.

This is the volume of the universe encompassed by redshifts less than z. For the case of omega_k = 0 it is a sphere of radius `comoving_distance(z)` but it is less intuitive if omega_k is not 0.

Parameters

z : array_like

Input redshifts.

Returns

V : ndarray, or float if input scalar

Comoving volume in Mpc^3 at each input redshift.

`Onu(z)`

Return the density parameter for massless neutrinos at redshift z.

Parameters

z : array_like

Input redshifts.

Returns

Onu : ndarray, or float if input scalar

The energy density of photons relative to the critical density at each redshift. Note that this includes only their relativistic energy, since they are assumed massless.

FlatLambdaCDM

class `astropy.cosmology.core.FlatLambdaCDM`(*H0*, *Om0*, *Tcmb0=2.725*, *Neff=3.04*, *name='FlatLambdaCDM'*)

Bases: `astropy.cosmology.core.LambdaCDM`

FLRW cosmology with a cosmological constant and no curvature.

This has no additional attributes beyond those of FLRW.

Examples

```
>>> from astro.cosmology import FlatLambdaCDM
>>> cosmo = FlatLambdaCDM(H0=70, Om0=0.3)
```

The comoving distance in Mpc at redshift z:

```
>>> dc = cosmo.comoving_distance(z)
```

Methods Summary

`efunc`(z)	Function used to calculate H(z), the Hubble parameter.
`inv_efunc`(z)	Function used to calculate $\frac{1}{H_z}$.

Methods Documentation

`efunc(z)`

Function used to calculate H(z), the Hubble parameter.

Parameters

z : array_like

Input redshifts.

Returns

E : ndarray, or float if input scalar

The redshift scaling of the Hubble consant.

Notes

The return value, E, is defined such that $H(z) = H_0 E$.

`inv_efunc(z)`

Function used to calculate $\frac{1}{H_z}$.

Parameters

z : array_like

Input redshifts.

Returns

E : ndarray, or float if input scalar

The inverse redshift scaling of the Hubble constant.

Notes

The return value, E, is defined such that $H_z = H_0 / E$.

Flatw0waCDM

class `astropy.cosmology.core.Flatw0waCDM`(*H0*, *Om0*, *w0=-1.0*, *wa=0.0*, *Tcmb0=2.725*, *Neff=3.04*, *name='Flatw0waCDM'*)

Bases: `astropy.cosmology.core.w0waCDM`

FLRW cosmology with a CPL dark energy equation of state and no curvature.

The equation for the dark energy equation of state uses the CPL form as described in Chevallier & Polarski Int. J. Mod. Phys. D10, 213 (2001) and Linder PRL 90, 91301 (2003): $w(z) = w_0 + w_a(1-a) = w_0 + w_a z/(1+z)$.

Examples

```
>>> from astro.cosmology import Flatw0waCDM
>>> cosmo = Flatw0waCDM(H0=70, Om0=0.3, w0=-0.9, wa=0.2)
```

The comoving distance in Mpc at redshift z:

```
>>> dc = cosmo.comoving_distance(z)
```

FlatwCDM

class `astropy.cosmology.core.FlatwCDM`(*H0*, *Om0*, *w0=-1.0*, *Tcmb0=2.725*, *Neff=3.04*, *name='FlatwCDM'*)

Bases: `astropy.cosmology.core.wCDM`

FLRW cosmology with a constant dark energy equation of state and no spatial curvature.

This has one additional attribute beyond those of FLRW.

Examples

```
>>> from astro.cosmology import FlatwCDM
>>> cosmo = FlatwCDM(H0=70, Om0=0.3, w0=-0.9)
```

The comoving distance in Mpc at redshift z:

```
>>> dc = cosmo.comoving_distance(z)
```

Methods Summary

efunc(z)	Function used to calculate H(z), the Hubble parameter.
inv_efunc(z)	Function used to calculate $\frac{1}{H_z}$.

Methods Documentation

efunc(z)
Function used to calculate H(z), the Hubble parameter.

Parameters
z : array_like

Input redshifts.

Returns
E : ndarray, or float if input scalar

The redshift scaling of the Hubble consant.

Notes

The return value, E, is defined such that $H(z) = H_0 E$.

inv_efunc(z)
Function used to calculate $\frac{1}{H_z}$.

Parameters
z : array_like

Input redshifts.

Returns
E : ndarray, or float if input scalar

The inverse redshift scaling of the Hubble constant.

Notes

The return value, E, is defined such that $H_z = H_0/E$.

LambdaCDM

class astropy.cosmology.core.LambdaCDM(*H0*, *Om0*, *Ode0*, *Tcmb0=2.725*, *Neff=3.04*, *name='LambdaCDM'*)

Bases: astropy.cosmology.core.FLRW

FLRW cosmology with a cosmological constant and curvature.

This has no additional attributes beyond those of FLRW.

Examples

```
>>> from astro.cosmology import LambdaCDM
>>> cosmo = LambdaCDM(H0=70, Om0=0.3, Ode0=0.7)
```

The comoving distance in Mpc at redshift z:

```
>>> dc = cosmo.comoving_distance(z)
```

Methods Summary

efunc(z)	Function used to calculate H(z), the Hubble parameter.
inv_efunc(z)	Function used to calculate $\frac{1}{H_z}$.
w(z)	Returns dark energy equation of state at redshift z.
de_density_scale(z)	Evaluates the redshift dependence of the dark energy density.

Methods Documentation

efunc(*z*)

Function used to calculate H(z), the Hubble parameter.

Parameters

z : array_like

Input redshifts.

Returns

E : ndarray, or float if input scalar

The redshift scaling of the Hubble consant.

Notes

The return value, E, is defined such that $H(z) = H_0 E$.

inv_efunc(*z*)

Function used to calculate $\frac{1}{H_z}$.

Parameters

z : array_like

Input redshifts.

Returns

E : ndarray, or float if input scalar

The inverse redshift scaling of the Hubble constant.

Notes

The return value, E, is defined such that $H_z = H_0/E$.

`w(z)`

Returns dark energy equation of state at redshift z.

Parameters

z : array_like

Input redshifts.

Returns

w : ndarray, or float if input scalar

The dark energy equation of state

Notes

The dark energy equation of state is defined as $w(z) = P(z)/\rho(z)$, where $P(z)$ is the pressure at redshift z and $\rho(z)$ is the density at redshift z, both in units where c=1. Here this is $w(z) = -1$.

`de_density_scale(z)`

Evaluates the redshift dependence of the dark energy density.

Parameters

z : array_like

Input redshifts.

Returns

I : ndarray, or float if input scalar

The scaling of the energy density of dark energy with redshift.

Notes

The scaling factor, I, is defined by $\rho(z) = \rho_0 I$, and in this case is given by $I = 1$.

w0waCDM

class `astropy.cosmology.core.w0waCDM`(*H0*, *Om0*, *Ode0*, *w0=-1.0*, *wa=0.0*, *Tcmb0=2.725*, *Neff=3.04*, *name='w0waCDM'*)

Bases: `astropy.cosmology.core.FLRW`

FLRW cosmology with a CPL dark energy equation of state and curvature.

The equation for the dark energy equation of state uses the CPL form as described in Chevallier & Polarski Int. J. Mod. Phys. D10, 213 (2001) and Linder PRL 90, 91301 (2003): $w(z) = w_0 + w_a(1-a) = w_0 + w_a z/(1+z)$.

Examples

```
>>> from astro.cosmology import w0waCDM
>>> cosmo = w0waCDM(H0=70, Om0=0.3, Ode0=0.7, w0=-0.9, wa=0.2)
```

The comoving distance in Mpc at redshift z:

```
>>> dc = cosmo.comoving_distance(z)
```

Attributes Summary

wa	Negative derivative of dark energy equation of state w.r.t.
w0	Dark energy equation of state at z=0

Methods Summary

w(z)	Returns dark energy equation of state at redshift z.
de_density_scale(z)	Evaluates the redshift dependence of the dark energy density.

Attributes Documentation

wa
: Negative derivative of dark energy equation of state w.r.t. a

w0
: Dark energy equation of state at z=0

Methods Documentation

w(*z*)
: Returns dark energy equation of state at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
w : ndarray, or float if input scalar

The dark energy equation of state

Notes

The dark energy equation of state is defined as $w(z) = P(z)/\rho(z)$, where $P(z)$ is the pressure at redshift z and $\rho(z)$ is the density at redshift z, both in units where c=1. Here this is $w(z) = w_0 + w_a(1-a) = w_0 + w_a \frac{z}{1+z}$.

de_density_scale(*z*)
: Evaluates the redshift dependence of the dark energy density.

Parameters
z : array_like

Input redshifts.

Returns
I : ndarray, or float if input scalar

The scaling of the energy density of dark energy with redshift.

Notes

The scaling factor, I, is defined by $\rho(z) = \rho_0 I$, and in this case is given by

$$I = (1+z)^{3(1+w_0+w_a)} \exp\left(-3w_a \frac{z}{1+z}\right)$$

w0wzCDM

class astropy.cosmology.core.w0wzCDM(*H0*, *Om0*, *Ode0*, *w0=-1.0*, *wz=0.0*, *Tcmb0=2.725*, *Neff=3.04*, *name='w0wzCDM'*)

Bases: astropy.cosmology.core.FLRW

FLRW cosmology with a variable dark energy equation of state and curvature.

The equation for the dark energy equation of state uses the simple form: $w(z) = w_0 + w_z z$.

This form is not recommended for z > 1.

Examples

```
>>> from astro.cosmology import wawzCDM
>>> cosmo = wawzCDM(H0=70, Om0=0.3, Ode0=0.7, w0=-0.9, wz=0.2)
```

The comoving distance in Mpc at redshift z:

```
>>> dc = cosmo.comoving_distance(z)
```

Attributes Summary

w0	Dark energy equation of state at z=0
wz	Derivative of the dark energy equation of state w.r.t.

Methods Summary

w(z)	Returns dark energy equation of state at redshift z.
de_density_scale(z)	Evaluates the redshift dependence of the dark energy density.

Attributes Documentation

w0

Dark energy equation of state at z=0

wz

Derivative of the dark energy equation of state w.r.t. z

Methods Documentation

w(*z*)
Returns dark energy equation of state at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
w : ndarray, or float if input scalar

The dark energy equation of state

Notes

The dark energy equation of state is defined as $w(z) = P(z)/\rho(z)$, where $P(z)$ is the pressure at redshift z and $\rho(z)$ is the density at redshift z, both in units where c=1. Here this is given by $w(z) = w_0 + w_z z$.

de_density_scale(*z*)
Evaluates the redshift dependence of the dark energy density.

Parameters
z : array_like

Input redshifts.

Returns
I : ndarray, or float if input scalar

The scaling of the energy density of dark energy with redshift.

Notes

The scaling factor, I, is defined by $\rho(z) = \rho_0 I$, and in this case is given by

$$I = (1+z)^{3(1+w_0-w_z)} \exp(-3w_z z)$$

wCDM
class astropy.cosmology.core.wCDM(*H0*, *Om0*, *Ode0*, *w0=-1.0*, *Tcmb0=2.725*, *Neff=3.04*, *name='wCDM'*)
Bases: astropy.cosmology.core.FLRW

FLRW cosmology with a constant dark energy equation of state and curvature.

This has one additional attribute beyond those of FLRW.

Examples

```
>>> from astro.cosmology import wCDM
>>> cosmo = wCDM(H0=70, Om0=0.3, Ode0=0.7, w0=-0.9)
```

The comoving distance in Mpc at redshift z:

```
>>> dc = cosmo.comoving_distance(z)
```

Attributes Summary

w0	Dark energy equation of state

Methods Summary

efunc(z)	Function used to calculate H(z), the Hubble parameter.
inv_efunc(z)	Function used to calculate $\frac{1}{H_z}$.
w(z)	Returns dark energy equation of state at redshift z.
de_density_scale(z)	Evaluates the redshift dependence of the dark energy density.

Attributes Documentation

w0
: Dark energy equation of state

Methods Documentation

efunc(z)
: Function used to calculate H(z), the Hubble parameter.

Parameters
z : array_like

Input redshifts.

Returns
E : ndarray, or float if input scalar

The redshift scaling of the Hubble consant.

Notes

The return value, E, is defined such that $H(z) = H_0 E$.

inv_efunc(z)
: Function used to calculate $\frac{1}{H_z}$.

Parameters
z : array_like

Input redshifts.

Returns
E : ndarray, or float if input scalar

The inverse redshift scaling of the Hubble constant.

Notes

The return value, E, is defined such that $H_z = H_0/E$.

w(*z*)
Returns dark energy equation of state at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
w : ndarray, or float if input scalar

The dark energy equation of state

Notes

The dark energy equation of state is defined as $w(z) = P(z)/\rho(z)$, where $P(z)$ is the pressure at redshift z and $\rho(z)$ is the density at redshift z, both in units where c=1. Here this is $w(z) = w_0$.

de_density_scale(*z*)
Evaluates the redshift dependence of the dark energy density.

Parameters
z : array_like

Input redshifts.

Returns
I : ndarray, or float if input scalar

The scaling of the energy density of dark energy with redshift.

Notes

The scaling factor, I, is defined by $\rho(z) = \rho_0 I$, and in this case is given by $I = (1 + z)^{3(1+w_0)}$

wpwaCDM
class astropy.cosmology.core.wpwaCDM(*H0*, *Om0*, *Ode0*, *wp=-1.0*, *wa=0.0*, *zp=0*, *Tcmb0=2.725*, *Neff=3.04*, *name='wpwaCDM'*)
Bases: astropy.cosmology.core.FLRW

FLRW cosmology with a CPL dark energy equation of state, a pivot redshift, and curvature.

The equation for the dark energy equation of state uses the CPL form as described in Chevallier & Polarski Int. J. Mod. Phys. D10, 213 (2001) and Linder PRL 90, 91301 (2003), but modified to have a pivot redshift as in the findings of the Dark Energy Task Force (Albrecht et al. arXiv:0901.0721 (2009)): $w(a) = w_p + w_a(a_p - a) = w_p + w_a(1/(1 + zp) - 1/(1 + z))$.

Examples

```
>>> from astro.cosmology import wpwaCDM
>>> cosmo = wpwaCDM(H0=70,Om0=0.3,Ode0=0.7,wp=-0.9,wa=0.2,zp=0.4)
```

The comoving distance in Mpc at redshift z:

```
>>> dc = cosmo.comoving_distance(z)
```

Attributes Summary

wa	Negative derivative of dark energy equation of state w.r.t.
wp	Dark energy equation of state at the pivot redshift zp
zp	The pivot redshift, where w(z) = wp

Methods Summary

w(z)	Returns dark energy equation of state at redshift z.
de_density_scale(z)	Evaluates the redshift dependence of the dark energy density.

Attributes Documentation

wa
: Negative derivative of dark energy equation of state w.r.t. a

wp
: Dark energy equation of state at the pivot redshift zp

zp
: The pivot redshift, where w(z) = wp

Methods Documentation

w(*z*)
: Returns dark energy equation of state at redshift z.

Parameters
z : array_like

Input redshifts.

Returns
w : ndarray, or float if input scalar

The dark energy equation of state

Notes

The dark energy equation of state is defined as $w(z) = P(z)/\rho(z)$, where $P(z)$ is the pressure at redshift z and $\rho(z)$ is the density at redshift z, both in units where c=1. Here this is $w(z) = w_p + w_a(a_p - a)$ where $a = 1/1 + z$ and $a_p = 1/1 + z_p$.

de_density_scale(*z*)
: Evaluates the redshift dependence of the dark energy density.

Parameters
z : array_like

Input redshifts.

Returns
I : ndarray, or float if input scalar

The scaling of the energy density of dark energy with redshift.

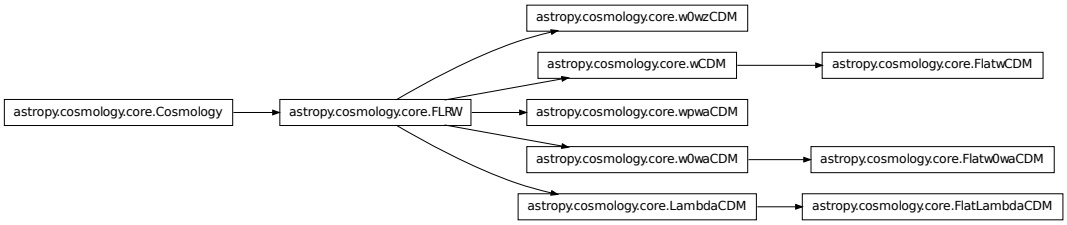
Notes

The scaling factor, I, is defined by $\rho(z) = \rho_0 I$, and in this case is given by

$$a_p = \frac{1}{1 + z_p}$$

$$I = (1 + z)^{3(1+w_p+a_p w_a)} \exp\left(-3w_a \frac{z}{1+z}\right)$$

Class Inheritance Diagram



1.10 FITS File handling (`astropy.io.fits`)

1.10.1 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.

1.10.2 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.

Reading and Updating Existing FITS Files

Opening a FITS file

Once the `astropy.io.fits` package is loaded using the standard convention[4], we can open an existing FITS file:

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

[4] For legacy code only that already depends on PyFITS, it's acceptable to continue using "from astropy.io import fits as pyfits".

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 Python list-like collection of HDU objects. An HDU (Header Data Unit) is the highest level component of the FITS file structure. So, after the above open call, hdulist[0] is the primary HDU, hdulist[1], if any, is the first extension HDU, etc. It should be noted that Astropy is using 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:

```
>>> hdulist.info()
Filename: test1.fits
No. Name  Type       Cards Dimensions Format
0 PRIMARY PrimaryHDU   220 ()         int16
1 SCI     ImageHDU      61 (800, 800) float32
2 SCI     ImageHDU      61 (800, 800) float32
3 SCI     ImageHDU      61 (800, 800) float32
4 SCI     ImageHDU      61 (800, 800) float32
```

After you are done with the opened file, close it with the HDUList.close() method:

```
>>> hdulist.close()
```

The headers will still be accessible after the HDUList is closed. The 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.

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 2-3 GB cannot be mmap'd (which is fine, because by that point you're likely to run out of physical memory anyways), but 64-bit systems are much less limited in this respect.

Working With a FITS Header

As mentioned earlier, each element of an HDUList is an HDU object with attributes of header and data, which can be used to access the header keywords and the data.

For those unfamiliar with FITS headers, they consist of a list of "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, float, or complex number. 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, simply do (a la Python dicts):

```
>>> hdulist[0].header['targname']
'NGC121'
```

to get the value of the keyword targname, which is a string 'NGC121'.

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 (a la Python lists):

```
>>> hdulist[0].header[27]
96
```

This example returns the 28th (like Python lists, it is 0-indexed) keyword's value–an integer–96.

Similarly, it is easy to update a keyword's value in Astropy, either through keyword name or index:

```
>>> prihdr = hdulist[0].header
>>> prihdr['targname'] = 'NGC121-a'
>>> prihdr[27] = 99
```

It is also possible to update both the value and comment associated with a keyword by assigning them as a tuple:

```
>>> prihdr = hdulist[0].header
>>> prihdr['targname'] = ('NGC121-a', 'the observation target')
>>> prihdr['targname']
'NGC121-a'
>>> prihdr.comments['targname']
'the observation target'
```

Like a dict, one 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's 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:

```
>>> prihdr.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:

```
>>> prihdr['history'] = 'I updated this file 2/26/09'
>>> prihdr['comment'] = 'Edwin Hubble really knew his stuff'
>>> prihdr['comment'] = 'I like using HST observations'
>>> prihdr['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 updating existing COMMENT or HISTORY cards, reference them by index:

```
>>> prihdr['history'][0] = 'I updated this file on 2/26/09'
>>> prihdr['history']
I updated this file on 2/26/09
```

To see the entire header as it appears in the FITS file (with the END card and padding stripped), simply enter the header object by itself, or print repr(header):

```
>>> header
SIMPLE  =                    T / file does conform to FITS standard
BITPIX  =                   16 / number of bits per data pixel
NAXIS   =                    0 / number of data axes
```

```
...all cards are shown...
>>> print repr(header)
...identical...
```

It's also possible to view a slice of the header:

```
>>> header[: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:

```
>>> prihdr.keys()
['SIMPLE', 'BITPIX', 'NAXIS', ...]
```

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.

```
>>> scidata = hdulist[1].data
```

Here, scidata points to the data object in the second HDU (the first HDU, `hdulist[0]`, being the primary HDU) in `hdulist`, which corresponds to the 'SCI' extension. Alternatively, you can access the extension by its extension name (specified in the EXTNAME keyword):

```
>>> scidata = hdulist['SCI'].data
```

If there is more than one extension with the same EXTNAME, EXTVER's value needs to be specified as the second argument, e.g.:

```
>>> scidata = hdulist['sci',2].data
```

The returned numpy object has many attributes and methods for a user to get information about the array, e. g.:

```
>>> scidata.shape
(800, 800)
>>> scidata.dtype.name
'float32'
```

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 scidata[1, 4]
```

Note that, like C (and unlike FORTRAN), Python is 0-indexed and the indices have the slowest axis first and fast axis last, i.e. for a 2-D image, the fast axis (X-axis) which corresponds to the FITS NAXIS1 keyword, is the second index. Similarly, the 1-indexed sub-section of x=11 to 20 (inclusive) and y=31 to 40 (inclusive) would be given in Python as:

```
>>> scidata[30:40, 10:20]
```

To update the value of a pixel or a sub-section:

```
>>> scidata[30:40, 10:20] = scidata[1, 4] = 999
```

This example changes the values of both the pixel [1, 4] and the sub-section [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 = hdulist[1].header['photflam']
>>> exptime = prihdr['exptime']
>>> scidata *= photflam / exptime
```

This example performs the math on the array in-place, thereby keeping the memory usage to a minimum.

If at this point you want to preserve all the changes you made and write it to a new file, you can use the `HDUList.writeto()` method (see below).

Working With Table Data

If you are familiar with the record array in numpy, 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.

Like images, the data portion of a FITS table extension is in the `.data` attribute:

```
>>> hdulist = fits.open('table.fits')
>>> tbdata = hdulist[1].data # assuming the first extension is a table
```

To see the first row of the table:

```
>>> print tbdata[0]
(1, 'abc', 3.7000002861022949, 0)
```

Each row in the table is a `FITS_rec` 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) of the table, use:

```
>>> tbdata.field(0)
array([1, 2])
```

A numpy object with the data type of the specified field is returned.

Like header keywords, a field can be referred either by index, as above, or by name:

```
>>> tbdata.field('id')
array([1, 2])
```

But how do we know what field names we've got? First, let's introduce another attribute of the table HDU: the `columns` attribute:

```
>>> cols = hdulist[1].columns
```

This attribute is a `ColDefs` (column definitions) object. If we use the `ColDefs.info()` method:

```
>>> cols.info()
 name:
      ['c1', 'c2', 'c3', 'c4']
 format:
      ['1J', '3A', '1E', '1L']
 unit:
      ['', '', '', '']
 null:
      [-2147483647, '', '', '']
 bscale:
      ['', '', 3, '']
 bzero:
      ['', '', 0.40000000000000002, '']
 disp:
      ['I11', 'A3', 'G15.7', 'L6']
 start:
      ['', '', '', '']
 dim:
      ['', '', '', '']
```

it will show all its attributes, such as names, formats, bscales, bzeros, etc. We can also get these properties individually, e.g.:

```
>>> cols.names
['ID', 'name', 'mag', 'flag']
```

returns a (Python) list of field names.

Since each field is a numpy object, we'll have the entire arsenal of numpy tools to use. We can reassign (update) the values:

```
>>> tbdata.field('flag')[:] = 0
```

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.

```
>>> hdulist.writeto('newimage.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 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.

```
>>> f = fits.open('original.fits', mode='update')
... # making changes in data and/or header
>>> f.flush() # changes are written back to original.fits
```

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 easy in Astropy for an image HDU. We'll first demonstrate how to create a FITS file consisting 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 a HDUList to contain the newly created primary HDU, and write to a new file:

```
>>> hdulist = fits.HDUList([hdu])
>>> hdulist.writeto('new.fits')
```

That's it! In fact, Astropy even provides a shortcut for the last two lines to accomplish the same behavior:

```
>>> hdu.writeto('new.fits')
```

Creating a New Table File

To create a table HDU is a little more involved than 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'll 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:

```
>>> from astropy.io import fits
>>> 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)
```

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 `new_table()` function:

```
>>> tbhdu = fits.new_table(cols)
```

This function returns (in this case) a BinTableHDU.

Of course, you can do this more concisely:

```
>>> from astropy.io import fits
>>> tbhdu = fits.new_table(
...     fits.ColDefs([fits.Column(name='target', format='20A', array=a1),
...                   fits.Column(name='V_mag', format='E', array=a2)]))
```

As before, we create a PrimaryHDU object to encapsulate the data:

```
>>> hdu = fits.PrimaryHDU(n)
```

We then create a HDUList containing both the primary HDU and the newly created table extension, and write to a new file:

```
>>> thdulist = fits.HDUList([hdu, tbhdu])
>>> thdulist.writeto('table.fits')
```

If this will be the only extension of the new FITS file and you only have a minimal primary HDU with no data, Astropy again provides a shortcut:

```
>>> tbhdu.writeto('table.fits')
```

Alternatively, you can append it to the hdulist we have already created from the image file section:

```
>>> hdulist.append(tbhdu)
```

So far, we have covered the most basic features of `astropy.io.fits`. In the following chapters we'll show more advanced examples and explain options in each class and method.

Convenience Functions

`astropy.io.fits` also provides several high level ("convenience") functions. Such a convenience function is a "canned" operation to achieve one simple task. By using these "convenience" functions, a user does not have to worry about opening or closing a file, all the housekeeping is done implicitly.

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 get:

```
>>> from astropy.io.fits import getheader
>>> getheader('in.fits') # get default HDU (=0), i.e. primary HDU's header
>>> getheader('in.fits', 0) # get primary HDU's header
>>> getheader('in.fits', 2) # the second extension
# the HDU with EXTNAME='sci' (if there is only 1)
>>> getheader('in.fits', 'sci')
# the HDU with EXTNAME='sci' and EXTVER=2
>>> getheader('in.fits', 'sci', 2)
>>> getheader('in.fits', ('sci', 2)) # use a tuple to do the same
>>> getheader('in.fits', ext=2) # the second extension
```

```
# the 'sci' extension, if there is only 1
>>> getheader('in.fits', extname='sci')
# the HDU with EXTNAME='sci' and EXTVER=2
>>> getheader('in.fits', extname='sci', extver=2)
# ambiguous specifications will raise an exception, DON'T DO IT!!
>>> getheader('in.fits', ext=('sci',1), extname='err', extver=2)
```

After you get the header, you can access the information in it, such as getting and modifying a keyword value:

```
>>> from astropy.io.fits import getheader
>>> hdr = getheader('in.fits', 1) # get first extension'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
>>> flt = getval('in.fits', 'filter', 1) # get 1st extension's keyword
                                         # FILTER's value
>>> val = getval('in.fits', 10, 'sci', 2) # get the 2nd sci extension's
                                          # 11th keyword's value
```

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
>>> dat = getdata('in.fits', 'sci', 3) # get 3rd sci extension's data
# get 1st extension's data and header
>>> data, hdr = getdata('in.fits', 1, header=True)
```

The functions introduced above are for reading. The next few functions demonstrate convenience functions for writing:

```
>>> fits.writeto('out.fits', data, header)
```

The `writeto()` function uses the provided data and an optional header to write to an output FITS file.

```
>>> fits.append('out.fits', data, header)
```

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(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
```

The update() function will update the specified extension with the input data/header. 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.

Finally, the info() function will print out information of the specified FITS file:

```
>>> fits.info('test0.fits')
Filename: test0.fits
No. Name    Type       Cards Dimensions Format
0   PRIMARY PrimaryHDU   138 ()         Int16
1   SCI     ImageHDU      61 (400, 400) Int16
2   SCI     ImageHDU      61 (400, 400) Int16
3   SCI     ImageHDU      61 (400, 400) Int16
4   SCI     ImageHDU      61 (400, 400) Int16
```

1.10.3 Using io.fits

FITS Headers

In the next three chapters, more detailed information as well as examples will be explained for manipulating the header, the image data, and the table data respectively.

Header of an HDU

Every HDU normally has two components: header and data. In Astropy these two components are accessed through the two attributes of the HDU, header and 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:

```
>>> 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 the required keywords from the input header first and then add back the required keywords compatible to the new HDU. 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 Quick Tutorial, keyword values can be accessed via keyword name or index of an HDU's header attribute. Here is a quick summary:

```
>>> hdulist = fits.open('input.fits') # open a FITS file
>>> prihdr = hdulist[0].header # the primary HDU header
>>> print prihdr[3] # get the 4th keyword's value
```

```
10
>>> prihdr[3] = 20 # change its value
>>> prihdr['DARKCORR']  # get the value of the keyword 'darkcorr'
'OMIT'
>>> prihdr['darkcorr'] = 'PERFORM'  # change darkcorr's value
```

Keyword names are case-insenstive except in a few special cases (see the sections on HIERARCH card and record-valued cards). Thus, `prihdr['abc']`, `prihdr['ABC']`, or `prihdr['aBc']` are all equivalent.

Like with python `dict`s, new keywords can also be added to the header using assignment syntax:

```
>>> 'DARKCORR' in header  # Check for existence
False
>>> header['DARKCORR'] = 'OMIT'  # Add a new DARKCORR keyword
```

You can also add a new value *and* comment by assigning them as a tuple:

```
>>> header['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:

  ```
  >>> header.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:

  ```
  >>> header.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 prihdr[3] # delete the 2nd keyword
>>> del prihdr['abc'] # get the value of the keyword 'abc'
```

Note that, like a regular Python list, the indexing updates after each delete, so if `del prihdr[3]` is done two times in a row, the 4th and 5th keywords are removed from the original header. Likewise, `del prihdr[-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 prihdr[3:5]
```

The method `Header.set()` is another way to update they 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 one to either move cards within the header, or specify the location of a new card relative to existing cards:

```
>>> prihdr.set('target', 'NGC1234', 'target name')
>>> # place the next new keyword before the 'target' keyword
>>> prihdr.set('newkey', 666, before='target') # comment is optional
>>> # place the next new keyword after the 21st keyword
>>> prihdr.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:

```
>>> header['NAXIS']
2
>>> header.comments['NAXIS']
the number of image axes
>>> header.comments['NAXIS'] = 'The number of image axes'  # Update
```

Comments can be accessed in all 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 `header.comments` instead of just `header` 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:

```
>>> prihdr['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 diplay just the last HISTORY entry, use `prihdr['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. Here is an example:

```
>>> 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.

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. But there's 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)`. For example:

```
>>> 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 card images
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.

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 non-standard 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; print c2
ABC = 3.456D023
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 it came from. Use the methods provided by the `Header` class instead.

CONTINUE Cards

The fact that the FITS standard only allows up to 8 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:

> http://legacy.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, even though it is not a FITS standard. The examples below show the use of CONTINUE is automatic for long string values.

```
>>> header = fits.Header()
>>> header['abc'] = 'abcdefg' * 20
>>> header
ABC = 'abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcd&'
CONTINUE 'efgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefga&'
CONTINUE 'bcdefg&'
>>> header['abc']
'abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefga
>>> # both value and comments are long
>>> header['abc'] = ('abcdefg' * 10, 'abcdefg' * 10)
>>> header
ABC = 'abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcd&'
CONTINUE 'efg&'
CONTINUE '&' / abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefga
CONTINUE '&' / bcdefg
```

Note that when a CONTINUE card is used, at the end of each 80-characters card image, an ampersand is present. The ampersand is not part of the string value. Also, there is no "=" at the 9th 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's 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 8 characters, there is a convention originated at 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 8 characters Astropy will automatically use a HIERARCH card, but will also issue a warning in case this is in error. However, one may explicitly request a HIERARCH card by prepending the keyword with 'HIERARCH ' (just as it would appear in the header). For example, `header['HIERARCH abcdefghi']` will create the keyword `abcdefghi` without displaying a warning. Once created, HIERARCH keywords can be accessed like any other: `header['abcdefghi']`, without prepending 'HIERARCH' to the keyword. HIEARARCH keywords also differ from normal FITS keywords in that they are case-sensitive.

Examples follow:

```
>>> c = fits.Card('abcdefghi', 10)
Keyword name 'abcdefghi' is greater than 8 characters; a HIERARCH card will
be created.
>>> print c
HIERARCH abcdefghi = 10
>>> c = fits.Card('hierarch abcdefghi', 10)
>>> print c
HIERARCH abcdefghi = 10
>>> h = fits.PrimaryHDU()
>>> h.header['hierarch abcdefghi'] =  99
>>> h.header['abcdefghi']
99
>>> h.header['abcdefghi'] = 10
>>> h.header['abcdefghi']
10
>>> h.header['ABCDEFGHI']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "astropy/io/fits/header.py", line 121, in __getitem__
    return self._cards[self._cardindex(key)].value
  File "astropy/io/fits/header.py", line 1106, in _cardindex
    raise KeyError("Keyword %r not found." % keyword)
KeyError: "Keyword 'ABCDEFGI.' not found."
>>> h.header
SIMPLE = T / conforms to FITS standard
BITPIX = 8 / array data type
NAXIS = 0 / number of array dimensions
EXTEND = T
HIERARCH abcdefghi = 1000
```

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 created CONTINUE and HIEARARCH cards. The Header is just a data structure, and as user you shouldn't have to worry about how it ultimately gets serialized to a header in a FITS file.

Though there are some areas where it's 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 where you have concern yourself unncessarily about how the header is constructed, then let help@stsci.edu know, as there are probably areas where this can be improved on even more.

Image Data

In this chapter, we'll 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. In Astropy, for most cases, it is just a simple numpy array, having the shape specified by the NAXIS keywords and the data type specified by the BITPIX keyword - unless the data is scaled, see 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
-32       numpy.float32
-64       numpy.float64
```

To recap the fact that 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).

Here is a summary of reading and updating image data values:

```
>>> f = fits.open('image.fits') # open a FITS file
>>> scidata = f[1].data # assume the first extension is an image
>>> print scidata[1,4] # get the pixel value at x=5, y=2
>>> scidata[30:40, 10:20] # get values of the subsection
                          # from x=11 to 20, y=31 to 40 (inclusive)
>>> scidata[1,4] = 999 # update a pixel value
>>> scidata[30:40, 10:20] = 0 # update values of a subsection
>>> scidata[3] = scidata[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 scidata to zero. The second one is to take logarithm of the pixel values which are positive:

```
>>> scidata[scidata<0] = 0
>>> scidata[scidata>0] = numpy.log(scidata[scidata>0])
```

These examples show the concise nature of numpy array operations.

Scaled Data

Sometimes an image is scaled, i.e. 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 scaled image is to store unsigned 16-bit integer data because 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 as specified by the BITPIX keyword. For 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). Here is an example of what happens to such a file, before and after the data is touched

```
>>> f = fits.open('scaled_uint16.fits')
>>> hdu = f[1]
>>> print hdu.header['bitpix'], hdu.header['bzero']
16 32768
>>> print hdu.data  # once data is touched, it is scaled
[ 11. 12. 13. 14. 15.]
>>> hdu.data.dtype.name
'float32'
>>> print hdu.header['bitpix']  # BITPIX is also updated
-32
# BZERO and BSCALE are removed after the scaling
>>> print hdu.header['bzero']
KeyError: "Keyword 'bzero' not found."
```

Warning: An important caveat to be aware of when dealing with scaled data in PyFITS, 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.
One may also manually reapply scale parameters by using `hdu.scale()` (see below). Alternately, one may open files with the `scale_back=True` argument. This assures that the original scaling is preserved when saving.

Writing Scaled Image Data With the extra processing and memory requirement, we discourage users to use scaled data as much as possible. However, Astropy does provide ways to write scaled data with the scale(type, option, bscale, bzero) method. Here are a few examples:

```
>>> # 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
>>> hdu.scale('int32', 'minmax')
>>> # scale the data, using the original BSCALE/BZERO
>>> hdu.scale('int32', 'old')
```

The first example above shows how to store an unsigned short integer array.

Great 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(numpy.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 images(s) ten rows at a time, the `section` attribute of an HDU can be used to alleviate such memory problems.

With PyFITS' 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 large than ~4 GB on a 32-bit system–in such cases it may be necessary to use sections.

Here is an example of getting the median image from 3 input images of the size 5000x5000:

```
>>> f1 = fits.open('file1.fits')
>>> f2 = fits.open('file2.fits')
>>> f3 = fits.open('file3.fits')
>>> output = numpy.zeros(5000 * 5000)
>>> for i in range(50):
... j = i * 100
... k = j + 100
... x1 = f1[1].section[j:k,:]
... x2 = f2[1].section[j:k,:]
... x3 = f3[1].section[j:k,:]
... # use scipy.stsci.image's median function
... output[j:k] = image.median([x1, x2, x3])
```

Data in each `section` does not need to be contiguous for memory savings to be possible. PyFITS will do its best to join together discontiguous sections of the array while reading as little as possible into memory.

Sections cannot be assigned to. Any modifications made to a data section are not saved back to the original file.

Table Data

In this chapter, we'll 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 table 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 takes up more storage space as well as more processing time since the ASCII text need to be parsed back into numerical values.

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 simple example of record array:

```
>>> from numpy import rec
>>> bright = 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 3 records (rows) and 4 fields (columns). The first field is a short integer, second a character string (of length 20), third a floating point number, and fourth a character string (of length 10). Each record has the same (heterogeneous) data structure.

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. To recap, the simple example in the Quick Tutorial:

```
>>> f = fits.open('bright_stars.fits') # open a FITS file
>>> tbdata = f[1].data # assume the first extension is a table
>>> print tbdata[:2] # show the first two rows
[(1, 'Sirius', -1.4500000476837158, 'A1V'),
(2, 'Canopus', -0.73000001907348633, 'F0Ib')]


>>> print tbdata.field('mag') # show the values in field "mag"
[-1.45000005 -0.73000002 -0.1 ]
>>> print tbdata.field(1) # field can be referred by index too
['Sirius' 'Canopus' 'Rigil Kent']
>>> scidata[1,4] = 999 # update a pixel value
>>> scidata[30:40, 10:20] = 0 # update values of a subsection
>>> scidata[3] = scidata[2] # copy the 3rd row to the 4th row
```

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, `tbdata.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 ommitted 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 containg 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.

In the next example, assuming the table's second field having the name 'magnitude', an output table containing all the records of magnitude > 5 from the input table is generated:

```
>>> from astropy.io import fits
>>> t = fits.open('table.fits')
>>> tbdata = t[1].data
>>> mask = tbdata.field('magnitude') > 5
>>> newtbdata = tbdata[mask]
>>> hdu = fits.BinTableHDU(newtbdata)
>>> hdu.writeto('newtable.fits')
```

Merging Tables Merging different tables is straightforward in Astropy. Simply merge the column definitions of the input tables:

```
>>> t1 = fits.open('table1.fits')
>>> t2 = fits.open('table2.fits')
# the column attribute is the column definitions
>>> t = t1[1].columns + t2[1].columns
>>> hdu = fits.new_table(t)
>>> hdu.writeto('newtable.fits')
```

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 don't 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.

Appending Tables Appending one table after another is slightly trickier, since the two tables may have different field attributes. Here are two examples. The first is to append by field indices, the second one is to append by field names. In both cases, the output table will inherit column attributes (name, format, etc.) of the first table.

```
>>> t1 = fits.open('table1.fits')
>>> t2 = fits.open('table2.fits')
# one way to find the number of records
>>> nrows1 = t1[1].data.shape[0]
# another way to find the number of records
>>> nrows2 = t2[1].header['naxis2']
# total number of rows in the table to be generated
>>> nrows = nrows1 + nrows2
>>> hdu = fits.new_table(t1[1].columns, nrows=nrows)
# first case, append by the order of fields
>>> for i in range(len(t1[1].columns)):
... hdu.data.field(i)[nrows1:]=t2[1].data.field(i)
# or, second case, append by the field names
>>> for name in t1[1].columns.names:
... hdu.data.field(name)[nrows1:]=t2[1].data.field(name)
# write the new table to a FITS file
>>> hdu.writeto('newtable.fits')
```

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 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 the scalings are done for the user, so the user only sees the physical data. Thus, this 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	4
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

We'll 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 in Column()	Corresponding header keyword	Description
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
array		the data of the column

Here are a few Columns using various combination of these arguments:

```
>>> import numpy as np
>>> from fits import Column
>>> counts = np.array([312, 334, 308, 317])
>>> names = np.array(['NGC1', 'NGC2', 'NGC3', 'NGC4'])
>>> c1 = Column(name='target', format='10A', array=names)
>>> c2 = Column(name='counts', format='J', unit='DN', array=counts)
```

```
>>> c3 = Column(name='notes', format='A10')
>>> c4 = Column(name='spectrum', format='1000E')
>>> c5 = Column(name='flag', format='L', array=[1, 0, 1, 1])
```

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 c4, each cell is an array (a numpy array) of 1000 elements.

For character string fields, the number can be either before or after the letter 'A' and they will mean the same string size. So, for columns c1 and c3, 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 `new_table()` function can be used to construct a table HDU. We can either go through the column definition object:

```
>>> coldefs = fits.ColDefs([c1, c2, c3, c4, c5])
>>> tbhdu = fits.new_table(coldefs)
```

or directly use the `new_table()` function:

```
>>> tbhdu = fits.new_table([c1, c2, c3, c4, c5])
```

A look of the newly created HDU's header will show that relevant keywords are properly populated:

```
>>> tbhdu.header
XTENSION = 'BINTABLE'                     / binary table extension
BITPIX   =                                8 / array data type
NAXIS    =                                2 / number of array dimensions
NAXIS1   =                             4025 / length of dimension 1
NAXIS2   =                                4 / length of dimension 2
PCOUNT   =                                0 / number of group parameters
GCOUNT   =                                1 / number of groups
TFIELDS  =                                5 / number of table fields
TTYPE1   = 'target '
TFORM1   = '10A '
TTYPE2   = 'counts '
TFORM2   = 'J '
TUNIT2   = 'DN '
TTYPE3   = 'notes '
TFORM3   = '10A '
TTYPE4   = 'spectrum'
TFORM4   = '1000E '
TTYPE5   = 'flag '
TFORM5   = 'L '
```

Warning: It should be noted that when creating a new table with `new_table()`, 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:

```
>>> c1 = Column(name='target', format='10A')
...
```

Then call `new_table()`:

```
>>> tbhdu = fits.new_table([c1, c2, c3, c4, c5])
```

This will create a new table HDU as before, with the correct column definitions, but an empty data section. Now simply assign your array directly to the HDU's data attribute:

```
>>> tbhdu.data = mydata
```

In a future version of Astropy table creation will be simplified and this process won't be necessary.

Verification

Astropy has built in a flexible scheme to verify FITS data being conforming to the FITS standard. The basic verification philosophy in Astropy is to be tolerant in input and strict in output.

When Astropy reads a FITS file which is not conforming 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 sub-standard (dialect?) and some of these become so prevalent that they 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 3 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 PyFITS 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 5 options for 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 5 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.

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 non-standard formatting is read and the user wants to copy or write out to an output file. The non-standard formatting will be preserved in the output file.
2. A user wants to create a non-standard 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.

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.

Verifications at Different Data Object Levels

We'll 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 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).
>>> h.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. Here is a lit of fixable and not fixable Cards:

Fixable Cards:

1. floating point numbers with lower case 'e' or 'd'
2. the equal sign is before column 9 in the card image
3. string value without enclosing quotes
4. missing equal sign before column 9 in the card image
5. space between numbers and E or D in floating point values
6. unparseable values will be "fixed" as a string

Here are some examples of fixable cards:

```
>>> hdu.header[4:] # has a bunch of fixable cards
FIX1 = 2.1e23
FIX2= 2
FIX3 = string value without quotes
FIX4 2
FIX5 = 2.4 e 03
FIX6 = '2 10 '
# can still access the values before the fix
>>> hdu.header[5]
2
>>> hdu.header['fix4']
2
>>> hdu.header['fix5']
2400.0
>>> hdu.verify('silentfix')
>>> hdu.header[4:]
FIX1 = 2.1E23
FIX2 = 2
FIX3 = 'string value without quotes'
FIX4 = 2
```

```
FIX5 = 2.4E03
FIX6 = '2 10 '
```

Unfixable Cards:

1. illegal characters in keyword name

We'll 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'")
# then append it to the header
>>> h.header.append(c)
# 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 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 a 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'.

Here are some examples:

```
>>> # 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'll 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 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.

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
>>> hdus = fits.open('ascii_table.fits')
>>> hdus[1].data[:1]
FITS_rec(
... [(10.123000144958496, 37)],
... dtype=[('a', '>f4'),('b','>i4')])
>>> hdus[1].data.field('a')
array([ 10.12300014, 5.19999981, 15.60999966, 0. ,
345. ], dtype=float32)
>>> hdus[1].data.formats
['E10.4', 'I5']
```

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 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        Single precision real
Ew.d        Single 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 '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 either `ColDef()` or `new_table()`.

The default value for tbtype is `BinTableHDU`.

```
>>>
# Define the columns
>>> import numpy as np
>>> from astropy.io import fits
>>> a1 = np.array(['abcd', 'def'])
>>> r1 = np.array([11., 12.])
>>> c1 = fits.Column(name='abc', format='A3', array=a1)
>>> c2 = fits.Column(name='def', format='E', array=r1,
...                  bscale=2.3, bzero=0.6)
>>> c3 = fits.Column(name='t1', format='I', array=[91, 92, 93])
```

```
# Create the table
>>> x = fits.ColDefs([c1, c2, c3], tbtype='TableHDU')
>>> hdu = fits.new_table(x, tbtype='TableHDU')
# Or, simply,
>>> hdu = fits.new_table([c1, c2, c3], tbtype='TableHDU')
>>> hdu.writeto('ascii.fits')
>>> hdu.data
FITS_rec([('abcd', 11.0, 91), ('def', 12.0, 92), ('', 0.0, 93)],
         dtype=[('abc', '|S3'), ('def', '|S14'), ('t1', '|S10')])
```

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 is 0, 1, or absent, t is one of the letter code for regular table data type (L, B, X, 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. So, for a variable length field of int32, The corresponding format spec is, e.g. 'PJ(100)'.

```
>>> f = fits.open('variable_length_table.fits')
>>> print f[1].header['tform5']
1PI(20)
>>> print f[1].data.field(4)[:3]
[array([1], dtype=int16) array([88, 2], dtype=int16)
array([ 1, 88, 3], dtype=int16)]
```

The above example shows a variable length array field of data type int16 and its first row has one element, second row has 2 elements etc. Accessing variable length fields is almost identical to regular fields, except that operations on the whole filed are usually not possible. A user has to process the field row by row.

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). Here is an example of creating a table with two fields, one is regular and the other variable length array.

```
>>> from astropy.io import fits
>>> import numpy as np
>>> c1 = fits.Column(name='var', format='PJ()',
...                  array=np.array([[45., 56]
                                     [11, 12, 13]],
...                                 dtype=np.object))
>>> c2 = fits.Column(name='xyz', format='2I', array=[[11, 3], [12, 4]])
# the rest is the same as a regular table.
# Create the table HDU
```

```
>>> tbhdu = fits.new_table([c1, c2])
>>> print tbhdu.data
FITS_rec([(array([45, 56]), array([11,  3], dtype=int16)),
          (array([11, 12, 13]), array([12,  4], dtype=int16))],
         dtype=[('var', '<i4', 2), ('xyz', '<i2', 2)])
# write to a FITS file
>>> tbhdu.writeto('var_table.fits')
>>> hdu = fits.open('var_table.fits')
# Note that heap info is taken care of (PCOUNT) when written to FITS file.
>>> hdu[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  =                   20 / number of group parameters
GCOUNT  =                    1 / number of groups
TFIELDS =                    2 / number of table fields
TTYPE1  = 'var '
TFORM1  = 'PJ(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, i.e. 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. Just use the .header attribute.

The content of the HDU can similarly be summarized by using the `HDUList.info()` method:

```
>>> f = fits.open('random_group.fits')
>>> print f[0].header['groups']
True
>>> print f[0].header['gcount']
7956
>>> print f[0].header['pcount']
6
>>> f.info()
Filename: random_group.fits
```

```
No. Name Type Cards Dimensions Format
0 AN GroupsHDU 158 (3, 4, 1, 1, 1) Float32 7956 Groups
6 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).

1. show the data in 100th group, including parameters and data

```
>>> print f[0].data[99]
(-8.1987486677035799e-06, 1.2010923615889215e-05,
-1.011189139244005e-05, 258.0, 2445728., 0.10, array([[[[[ 12.4308672 ,
0.56860745, 3.99993873],
[ 12.74043655, 0.31398511, 3.99993873],
[ 0. , 0. , 3.99993873],
[ 0. , 0. , 3.99993873]]]]], dtype=float32))
```

The data first lists all 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:

```
>>> f[0].data.parnames # get the parameter names
['uu--', 'vv--', 'ww--', 'baseline', 'date', 'date']
```

The group parameter can be accessed by the `par()` method. Like the table `field()` method, the argument can be either index or name:

```
>>> print f[0].data.par(0)[99] # Access group parameter by name or by index
-8.1987486677035799e-06
>>> print f[0].data.par('uu--')[99]
-8.1987486677035799e-06
```

Note that the parameter name 'date' appears twice. This is a feature in the random access group, and it means to add the values together. Thus:

```
>>>
# Duplicate group parameter name 'date' for 5th and 6th parameters
>>> print f[0].data.par(4)[99]
2445728.0
>>> print f[0].data.par(5)[99]
0.10
# When accessed by name, it adds the values together if the name is shared
# by more than one parameter
>>> print f[0].data.par('date')[99]
2445728.10
```

The :meth'~GroupData.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):

```
>>>
# Access group parameter by selecting the row (group) number last
>>> print f[0].data.par(0)[99]
-8.1987486677035799e-06
```

```
# Access group parameter by selecting the row (group) number first
>>> print f[0].data[99].par(0)
-8.1987486677035799e-06
```

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
>>> f[0].data.par(0)[99] = 99.
>>>
# Update group parameter when selecting the row (group) number first
>>> f[0].data[99].setpar(0, 99.) # or setpar('uu--', 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
>>> f[0].data[99].setpar('date', (2445729., 0.3))
>>> f[0].data[:3].setpar('date', (2445729., [0.11, 0.22, 0.33]))
>>> f[0].data[:3].par('date')
array([ 2445729.11 , 2445729.22 , 2445729.33000001])
```

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 f[0].data.data[99]
array([[[[[ 12.4308672 , 0.56860745, 3.99993873],
[ 12.74043655, 0.31398511, 3.99993873],
[ 0. , 0. , 3.99993873],
[ 0. , 0. , 3.99993873]]]]], type=float32)
```

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:

```
>>>
# Create the image arrays. The first dimension is the number of groups.
>>> imdata = numpy.arange(100.0, shape=(10, 1, 1, 2, 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 = numpy.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, parnames=['abc', 'xyz'],
...                    pardata=[pdata1, pdata2], bitpix=-32)
# 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 =            2
NAXIS4 =            1
NAXIS5 =            1
EXTEND =            T
GROUPS =            T / has groups
PCOUNT =            2 / number of parameters
GCOUNT =           10 / number of groups
PTYPE1 = 'abc '
PTYPE2 = 'xyz '
>>> print hdu.data[:2]
FITS_rec[
(0.10000000149011612, 42.0, array([[[[ 0., 1., 2., 3., 4.],
[ 5., 6., 7., 8., 9.]]]], dtype=float32)),
(1.1000000238418579, 42.0, array([[[[ 10., 11., 12., 13., 14.],
[ 15., 16., 17., 18., 19.]]]], dtype=float32))
]
```

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:

> http://www.adass.org/adass/proceedings/adass99/P2-42/

and "Registered FITS Convention, Tiled Image Compression Convention":

> http://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.

The content of the HDU header may be accessed using the `.header` attribute:

```
>>> f = fits.open('compressed_image.fits')
>>> print f[1].header
XTENSION= 'IMAGE   '           / extension type
BITPIX  =                   16 / array data type
NAXIS   =                    2 / number of array dimensions
NAXIS1  =                  512 / length of data axis
```

```
NAXIS2  =                  512 / length of data axis
PCOUNT  =                    0 / number of parameters
GCOUNT  =                    1 / one data group (required keyword)
EXTNAME = 'COMPRESSED'         / name of this binary table extension
```

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).

```
>>> f = fits.open('compressed_image.fits')
>>> print f[1]._header
XTENSION= 'BINTABLE'           / binary table extension
BITPIX  =                    8 / 8-bit bytes
NAXIS   =                    2 / 2-dimensional binary table
NAXIS1  =                    8 / width of table in bytes
NAXIS2  =                  512 / number of rows in table
PCOUNT  =               157260 / size of special data area
GCOUNT  =                    1 / one data group (required keyword)
TFIELDS =                    1 / number of fields in each row
TTYPE1  = 'COMPRESSED_DATA'    / label for field   1
TFORM1  = '1PB(384)'           / data format of field: variable length array
ZIMAGE  =                    T / extension contains compressed image
ZBITPIX =                   16 / data type of original image
ZNAXIS  =                    2 / dimension of original image
ZNAXIS1 =                  512 / length of original image axis
ZNAXIS2 =                  512 / length of original image axis
ZTILE1  =                  512 / 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
EXTNAME = 'COMPRESSED'         / 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('compressed_image.fits')
Filename: compressed_image.fits
No.    Name         Type      Cards   Dimensions   Format
0    PRIMARY     PrimaryHDU       6  ()            int16
1    COMPRESSED  CompImageHDU    52  (512, 512)    int16
>>>
>>> f = fits.open('compressed_image.fits')
>>> f.info()
Filename: compressed_image.fits
No.    Name         Type      Cards   Dimensions   Format
0    PRIMARY     PrimaryHDU       6  ()            int16
1    COMPRESSED  CompImageHDU    52  (512, 512)    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.

The content of the HDU data may be accessed using the `.data` attribute:

```
>>> f = fits.open('compressed_image.fits')
>>> f[1].data
array([[38, 43, 35, ..., 45, 43, 41],
       [36, 41, 37, ..., 42, 41, 39],
       [38, 45, 37, ..., 42, 35, 43],
       ...,
       [49, 52, 49, ..., 41, 35, 39],
       [57, 52, 49, ..., 40, 41, 43],
       [53, 57, 57, ..., 39, 35, 45]], dtype=int16)
```

Creating a Compressed Image HDU 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 other image HDU.

```
>>> 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.

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. Verify and update checksums, tolerating non-standard checksums, updating to standard checksum:

   ```
   $ fitscheck --checksum either --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. Verify original nonstandard checksums only:

   ```
   $ fitscheck --checksum nonstandard *.fits
   ```

5. Only check and fix compliance problems, ignoring checksums:

   ```
   $ fitscheck --checksum none --compliance --write *.fits
   ```

6. Verify standard interoperable checksums:

   ```
   $ fitscheck *.fits
   ```

7. Delete checksum keywords:

   ```
   $ fitscheck --checksum none --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 `fitscheck --help` to see the full program usage documentation.

Miscellaneous Features

In this chapter, we'll describe some of the miscellaneous features of Astropy.

Warning Messages

Astropy uses the Python warnings module to issue warning messages. 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 writeto convenience function may be suppressed from within a python script as follows:

```
import warnings
from astropy.io import fits

# ...

warnings.resetwarnings()
warnings.filterwarnings('ignore', category=UserWarning, append=True)
fits.writeto(file, im, clobber=True)
warnings.resetwarnings()
warnings.filterwarnings('always', category=UserWarning, append=True)
```

```
# ...
```

Astropy also issues warnings when deprecated API features are used. In Python 2.7 and up deprecation warnings are ignored by default. To run Python with deprecation warnings enabled, either start Python with the `-Wall` argument, or you can enable deprecation warnings specifically with `-Wd`.

In Python versions below 2.7, if you wish to *squelch* deprecation warnings, you can start Python with `-Wi::Deprecation`. This sets all deprecation warnings to ignored. See http://docs.python.org/using/cmdline.html#cmdoption-unittest-discover-W for more information on the -W argument.

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. See the API documentation for details on the different differ classes.

Examples

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

Figure 1.1: Red color information

Figure 1.2: Green color information

Figure 1.3: Blue color information

```
#!/usr/bin/env python
import numpy
import Image

from astropy.io import fits

#get the image and color information
image = Image.open('hs-2009-14-a-web.jpg')
#image.show()
xsize, ysize = image.size
r, g, b = image.split()
rdata = r.getdata() # data is now an array of length ysize\*xsize
gdata = g.getdata()
bdata = b.getdata()

# create numpy arrays
npr = numpy.reshape(rdata, (ysize, xsize))
npg = numpy.reshape(gdata, (ysize, xsize))
npb = numpy.reshape(bdata, (ysize, xsize))

# write out the fits images, the data numbers are still JUST the RGB
# scalings; don't use for science
red = fits.PrimaryHDU(data=npr)
red.header['LATOBS'] = "32:11:56" # add spurious header info
red.header['LONGOBS'] = "110:56"
red.writeto('red.fits')

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

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

1.10.4 Other Information

PyFITS FAQ

Contents

General Questions

What is PyFITS? 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 non-standard formats found in some FITS files.

PyFITS includes two command-line utilities for working with FITS files: fitscheck, which can verify and write FITS checksums; and fitsdiff, which can analyze and display the differences between two FITS files.

Although PyFITS is written mostly in Python, it includes an optional module written in C that's 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 is 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 (see LICENSE.txt in the PyFITS source code).

PyFITS' current primary developer and active maintainer is Erik Bray, though patch submissions are welcome from anyone. It has a Trac site where the source code can be browsed, and where bug reports may be submitted. The source code resides primarily in an SVN repository which allows anonymous checkouts, though a Git mirror also exists. PyFITS also has a GitHub site.

The current stable release series is 3.0.x. Each 3.0.x release tries to contain only bug fixes, and to not introduce any significant behavioral or API changes (though this isn't guaranteed to be perfect). The upcoming 3.1 release will contain new features and some API changes, though will try maintain as much backwards-compatibility as possible.

After the 3.1 release there may be further 3.0.x releases for bug fixes only where possible. Older versions of PyFITS (2.4 and earlier) are no longer actively supported.

PyFITS is also included as a major component of upcoming Astropy project as the `astropy.io.fits` module. The goal is for Astropy to eventually serve as a drop-in replacement for PyFITS (it even includes a legacy-compatibility mode where the `astropy.io.fits` module can still be imported as `pyfits`. However, for the time being PyFITS will still be released as an independent product as well, until such time that the Astropy project proves successful and widely-adopted.

Build and Installation Questions

Is PyFITS available on Windows? Yes–the majority of PyFITS is pure Python, and can be installed and used on any platform that supports Python (>=2.5). However, PyFITS includes an optional C extension module for reading/writing compressed image HDUs. As most Windows systems are not configured to compile C source code, binary installers are also available for Windows. Though PyFITS can also be installed from source even on Windows systems without a compiler by disabling the compression module. See How do I install PyFITS from source on Windows? for more details.

Where is the Windows installer for version X of PyFITS on version Y of Python? Every official PyFITS build for Windows is eventually uploaded to PyPI. This includes builds for every major Python release from 2.5.x and up, except for 3.0 as there is no official Numpy release for Python 3.0 on Windows. The one binary module included in these builds was linked with Numpy 1.6.1, though it should work with other recent Numpy versions.

Sometimes the Windows binary installers don't go up immediately after every PyFITS release. But if they appear missing they should go up within another business day or two. This has gotten better with recent releases thanks to some automation.

Why is the PyFITS installation failing on Windows? The most likely cause of installation failure on Windows is if building/ installing from source fails due to the lack of a compiler for the optional C extension module. Such a failure would produce an error that looks something like:

```
building 'pyfits.compression' extension
error: Unable to find vcvarsall.bat
```

Your best bet in cases like this is to install from one of the binary executable installers available for Windows on PyPI. However, there are still cases where you may need to install from source: For example, it's difficult to use the binary installers with virtualenv. See How do I install PyFITS from source on Windows? for more detailed instructions on building on Windows.

For other installation errors not mentioned by this FAQ, please contact help@stsci.edu with a description of the problem.

How do I install PyFITS from source on Windows? There are a few options for building/installing PyFITS from source on Windows.

First of all, as mentioned elsewhere, most of PyFITS is pure-Python. Only the C extension module for reading/writing compressed images needs to be compiled. If you don't need compressed image support, PyFITS can be installed without it.

In future releases this will hopefully be even easier, but for now it's necessary to edit one file in order to disable the extension module. Locate the setup.cfg file at the root of the PyFITS source code. This is the file that describes what needs to be installed for PyFITS. Find the line that reads `[extension=pyfits.compression]`. This is the section that lists what needs to be compiled for the extension module. Comment out every line in the extension section by prepending it with a # character (stopping at the `[build_ext]` line). It should look like this:

```
...
scripts = scripts/fitscheck

#[extension=pyfits.compression]
#sources =
#    src/compress.c
#    src/fits_hcompress.c
#    src/fits_hdecompress.c
#    src/fitsio.c
#    src/pliocomp.c
#    src/compressionmodule.c
#    src/quantize.c
#    src/ricecomp.c
#    src/zlib.c
#    src/inffast.c
#    src/inftrees.c
#    src/trees.c
#include_dirs = numpy
# Squelch a handful of warnings (which actually cause pip to break in tox and
# other environments due to gcc outputting non-ASCII characters in some
# terminals; see python issue6135)
#extra_compile_args =
#    -Wno-unused-function
#    -Wno-strict-prototypes

[build_ext]
...
```

With these lines properly commented out, rerun `python setup.py install`, and it should skip building/installing the compression module. PyFITS will work fine with out it, but will issue warnings when encountering a compressed image that it can't read.

If you do need to compile the compression module, this can still be done on Windows with just a little extra work. By default, Python tries to compile extension modules with the same compiler that Python itself was compiled with.

To check what compiler Python was built with, the easiest way is to run:

```
python -c "import platform; print platform.python_compiler()"
```

For the official builds of recent Python versions this should be something like:

```
MSC v.1500 32 bit (Intel)
```

For unofficial Windows distributions of Python, such as ActiveState, EPD, or Cygwin, your mileage may vary.

As it so happens, MSC v.15xx is the compiler version included with Visual C++ 2008. Luckily, Microsoft distributes a free version of this as Visual C++ Express Edition. So for building Python extension modules on Windows this is one of the simpler routes. Just install the free VC++ 2008. It should install a link to the Start Menu at All Programs->Microsoft Visual C++ Express Edition->Visual Studio Tools->Visual Studio 2008 Command Prompt.

If you run that link, it should launch a command prompt with reasonable environment variables set up for using Visual C++. Then change directories to your copy of the PyFITS source code and re-run `python setup.py install`. You may also need to comment out the `extra_compile_args` option in the `setup.cfg` file (its value is the two lines under it after the equal sign). Though the need to manually disable this option for MSC will be fixed in a future PyFITS version.

Another option is to use gcc through MinGW, which is in fact how the PyFITS releases for Windows are currently built. This article provides a good overview of how to set this up: http://seewhatever.de/blog/?p=217

Is PyFITS available for Mac OSX? Yes, but there is no binary package specifically for OSX (such as a .dmg, for example). For OSX just download, build, and install the source package. This is generally easier on OSX than it is on Windows, thanks to the more developer-friendly environment.

The only major problem with building on OSX seems to occur for some users of 10.7 Lion, with misconfigured systems. See the next question for details on that.

To build PyFITS without the optional compression module, follow the instructions in How do I install PyFITS from source on Windows?.

Why is the PyFITS installation failing on OSX Lion (10.7)? There is a common problem that affects all Python packages with C extension modules (not just PyFITS) for some users of OSX 10.7. What usually occurs is that when building the package several errors will be output, ending with something like:

```
unable to execute gcc-4.2: No such file or directory
error: command 'gcc-4.2' failed with exit status 1
```

There are a few other errors like it that can occur. The problem is that when you build a C extension, by default it will try to use the same compiler that your Python was built with. In this case, since you're using the 32-bit universal build of Python it's trying to use the older gcc-4.2 and is trying to build with PPC support, which is no longer supported in Xcode.

In this case the best solution is to install the x86-64 build of Python for OSX (http://www.python.org/ftp/python/2.7.2/python-2.7.2-macosx10.6.dmg for 2.7.2). In fact, this is the build version officially supported for use on Lion. Other, unofficial Python builds such as from MacPorts may also work.

How do I find out what version of PyFITS I have installed? To output the PyFITS version from the command line, run:

```
$ python -c 'import pyfits; print(pyfits.__version__)'
```

When PyFITS is installed with stsci_python, it is also possible to check the installed SVN revision by importing `pyfits.svn_version`. Then use `dir(pyfits.svn_version)` to view a list of available attributes. A feature like this will be available soon in standalone versions of PyFITS as well.

How do I run the tests for PyFITS? Currently the best way to run the PyFITS tests is to download the source code, either from a source release or from version control, and to run the tests out of the source. It is not necessary to install PyFITS to run the tests out of the source code.

The PyFITS tests require nose to run. nose can be installed on any Python version using pip or easy_install. See the nose documentation for more details.

With nose installed, it is simple to run the tests on Python 2.x:

```
$ python setup.py nosetests
```

If PyFITS has not already been built, this will build it automatically, then run the tests. This does not cause PyFITS to be installed.

On Python 3.x the situation is a little more complicated. This is due to the fact that PyFITS' source code is not Python 3-compatible out of the box, but has to be run through the 2to3 converter. Normally when you build/install PyFITS on Python 3.x, the 2to3 conversion is performed automatically. Unfortunately, nose does not know to use the 2to3'd source code, and will instead try to import and test the unconverted source code.

To work around this, it is necessary to first build PyFITS (which will run the source through 2to3):

```
$ python setup.py build
```

Then run the `nosetests` command, but pointing it to the `build` tree where the 2to3'd source code and tests reside, using the `-w` switch:

```
$ python setup.py nosetests -w build/lib.linux-x86_64-3.2
```

where the exact path of the `build/lib.*` directory will vary depending on your platform and Python version.

How can I build a copy of the PyFITS documentation for my own use? First of all, it's worth pointing out that the documentation for the latest version of PyFITS can always be downloaded in PDF form or browsed online in HTML. There are also plans to make the docs for older versions of PyFITS, as well as up-to-date development docs available online.

Otherwise, to build your own version of the docs either for offline use, or to build the development version of the docs there are a few reqirements. The most import requirement is Sphinx, which is the toolkit used to generate the documentation. Use `pip install sphinx` or `easy_install sphinx` to install Sphinx. Using pip or easy_install will install the correct versions of Sphinx's basic dependencies, which include docutils, Jinja2, and Pygments.

Next, the docs require STScI's custom Sphinx theme, stsci.sphinxext. It's a simple pure-Python pacakge and can be installed with pip or easy_install.

The next requirement is numpydoc, which is not normally installed with Numpy itself. Install it with pip or easy_install. Numpy is also required, though it is of course a requirement of PyFITS itself.

Finally, it is necessary to have matplotlib, specifically for matplotlib.sphinxext. This is perhaps the most onerous requirement if you do not already have it instaled. Please refer to the matplotlib documentation for details on downloading and installing matplotlib.

It is also necessary to install PyFITS itself in order to generate the API documentation. For this reason, it is a good idea to install Sphinx and PyFITS into a virtualenv in order to build the development version of the docs (see below).

With all the requirements installed, change directories into the `docs/` directory in the PyFITS source code, and run:

```
$ make html
```

to build the HTML docs, which will be output to `build/html`. To build the docs in other formats, please refer to the Sphinx documentation.

To summarize, assuming that you already have Numpy and Matplotlib on your Python installation, perform the following steps from within the PyFITS source code:

```
$ virtualenv --system-site-packages pyfits-docs
$ source pyfits-docs/bin/activate
$ pip install sphinx
$ pip install numpydoc
$ pip install stsci.sphinxext
$ python setup.py install pyfits
$ cd docs/
$ make html
```

Usage Questions

Something didn't 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. PyFITS 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, PyFITS should be very stable and well-tested. Before every PyFITS release it is ensured that all its tests pass on a variety of platforms, and those tests cover the majority of use-cases (until new corner cases are discovered).

PyFITS crashed and output a long string of code. What do I do? This listing of code is what is knows 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 PyFITS is meant to be used as a piece in other software projects, some exceptions raised by PyFITS 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 non-existent keyword in a header:

```
>>> import pyfits
>>> h = pyfits.Header()
>>> h['NAXIS']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/path/to/pyfits/header.py", line 125, in __getitem__
    return self._cards[self._cardindex(key)].value
  File "/path/to/pyfits/header.py", line 1535, in _cardindex
    raise KeyError("Keyword %r not found." % keyword)
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 PyFITS, it may indicate a bug in that software, in that it expected to find a keyword that didn't 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 PyFITS. So if you're getting an exception and you really don't 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 PyFITS? As mentioned elsewhere in this FAQ, there are many unusual corner cases when dealing with FITS files. It's possible that a file should work, but isn't support due to a bug. Sometimes it's even possible for a file to work in an older version of PyFITS, but not a newer version due to a regression that isn't 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 common-place that it is necessary to support them in any FITS readers. CONTINUE cards are one such example. There are non-standard conventions supported by PyFITS that are not supported by CFITSIO and vice-versa. You may have hit one of those cases.

If PyFITS is having trouble opening a file, a good way to rule out whether not the problem is with PyFITS is to run the file through the fitsverify. 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 PyFITS is still having trouble opening it (especially if it produces a traceback) then it's likely there is a bug in PyFITS.

How do I turn off the warning messages PyFITS keeps outputting to my console? PyFITS uses Python's built-in warnings subsystem for informating about exceptional conditions in the code that are recoverable, but that the user may way to be informed of. One of the most common warnings in PyFITS 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 PyFITS can be assumed to be from the warnings subsystem. Fortunately there are two easy ways to quiet these warnings:

1. Using the -W option to the `python` executable. Just start Python like:

   ```
   $ python -Wignore <scriptname>
   ```

   or for short:

   ```
   $ python -Wi <scriptname>
   ```

   and all warning output will be silenced.

2. Warnings can be silenced programatically from anywhere within a script. For example, to disable all warnings in a script, add something like:

   ```
   import warnings
   warnings.filterwarnings('ignore')
   ```

Another option, instead of `ignore` is `once`, which causes any warning to be output only once within the session, rather than repeatedly (such as in a loop). There are many more ways to filter warnings with `-W` and the warnings module. For example, it is possible to silence only specific warning messages. Please refer to the Python documentation for more details, or ask at help@stsci.edu.

How can I check if my code is using deprecated PyFITS features? PyFITS 3.0 included a major reworking of the code and some of the APIs. Most of the differences are just renaming functions to use a more consistent naming scheme. For example the `createCard()` function was renamed to `create_card()` for consistency with a `lower_case_underscore` naming scheme for functions.

There are a few other functions and attributes that were deprecated either because they were renamed to something simpler or more consistent, or because they were redundant or replaced.

Eventually all deprecated features will be removed in future PyFITS versions (though there will be significant warningsin advance). It is best to check whether your code is using deprecatd features sooner rather than later.

On Python 2.5, all deprecation warnigns are displayed by default, so you may have already discovered them. However, on Python 2.6 and up, deprecation warnings are *not* displayed by default. To show all deprecation warnings, start Python like:

```
$ python -Wd <scriptname>
```

Most deprecation issues can be fixed with a simple find/replace. The warnings displayed will let you know how to replace the old interface.

If you have a lot of old code that was written for older versions of PyFITS it would be worth doing this. PyFITS 3.1 introduces a significant rewrite of the Header interface, and contains even more deprecations.

What convention does PyFITS use for indexing, such as of image coordinates? All arrays and sequences in PyFITS 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 PyFITS 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 PyFITS is likely to cause more difficulties than it's worth.

How do I open a very large image that won't fit in memory? Prior to PyFITS 3.1, when the data portion of an HDU is accessed, the data is read into memory in its entirety. For example:

```
>>> hdul = pyfits.open('myimage.fits')
>>> hdul[0].data
...
```

reads the entire image array from disk into memory. For very large images or tables this is clearly undesirable, if not impossible given the available resources.

However, `pyfits.open()` has an option to access the data portion of an HDU by memory mapping using mmap. 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 I request just a slice of the image, such as `hdul[0].data[100:200]`, then just 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 use memory mapping, just add the `memmap=True` argument to `pyfits.open()`.

In PyFITS 3.1, the mmap support is improved enough that `memmap=True` is the default for all `pyfits.open()` calls. The default can also be controlled through an environment variable called `PYFITS_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 memmap, 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 flakey with sections too, though that will be fixed. See the PyFITS documentation for more details on working with sections.

How can I create a very large FITS file from scratch? This is a very common issue, but unfortunately PyFITS does not come with any built-in facilities for creating large files (larger than will fit in memory) from scratch (though it may in the future).

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

```
>> data = numpy.zeros((40000, 40000), dtype=numpy.float64)
>> hdu = pyfits.PrimaryHDU(data=data)
>> 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 anticpate 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 PyFITS will add automatically, create the header and pad it out to 36 * 4 cards like so:

```
>>> data = numpy.zeros((100, 100), dtype=numpy.float64)
# This is a stub array that we'll be using the initialize the HDU; its
# exact size is irrelevant, as long as it has the desired number of
# dimensions
>>> hdu = pyfits.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 PyFITS to "helpfully" reset the NAXISn keywords to match the size of the dummy array:

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

Finally, we need to 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:
...     fobj.seek(len(header.tostring()) + (40000 * 40000 * 8) - 1)
...     # The -1 is to account for the final byte that we are about to
...     # write
...     fobj.write('\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 HFS+) 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.

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 PyFITS is 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.

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? When you open a FITS file with `pyfits.open()`, a `pyfits.HDUList` object is returned, which holds all the HDUs in the file. This `HDUList` class is a subclass of Python's builtin `list`, and can be created from scratch and used as such:

```
>>> new_hdul = pyfits.HDUList()
>>> new_hdul.append(pyfits.ImageHDU())
>>> new_hdul.append(pyfits.ImageHDU())
>>> new_hdul.writeto('test.fits')
```

That will create a new multi-extension FITS file with two empty IMAGE extensions (a default PRIMARY HDU is prepended automatically if one was not provided manually).

Why is an image containing integer data being converted unexpectedly to floats? If the header for your image contains non-trivial 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 of guard if you're not expecting it, because it doesn't happen until the data portion of the HDU is accessed (to allow things like updating the header without rescaling the data). For example:

```
>>> hdul = pyfits.open('scaled.fits')
>>> image = hdul['SCI', 1]
>>> image.header['BITPIX']
32
>>> image.header['BSCALE']
2.0
>>> data = image.data  # Read the data into memory
>>> data.dtype
dtype('float64')  # Got float64 despite BITPIX = 32 (32-bit int)
>>> image.header['BITPIX']  # The BITPIX will automatically update too
-64
>>> '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
```

To prevent rescaling from occurring at all (good for updating headers–even if you don't intend for the code to access the data, it's good to err on the side of caution here), use the `do_not_scale_image_data` argument when opening the file:

```
>>> hdul = pyfits.open('scaled.fits', do_not_scale_image_data=True)
>>> image = hdul['SCI', 1]
>>> image.data.dtype
dtype('int32')
```

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 PyFITS 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:

```
>>> c = pyfits.Card.fromstring('FOO     = 1234567890.123456789')
>>> h = pyfits.Header()
>>> h.append(c)
>>> h
FOO     = 1234567890.123456789
```

As long as you don't assign new values to 'FOO' via `h['FOO'] = 123`, PyFITS will maintain the header value exactly as you formatted it.

astropy.io.fits History

Prior to its inclusion in Astropy, the `astropy.io.fits` package was a stand- alone package called PyFITS. Though for the time being active development is continuing on PyFITS, that development is also being merged into Astropy. This page documents the release history of PyFITS prior to its merge into Astropy.

PyFITS Changelog

3.2 (unreleased)

- 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 v3.30, 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)

3.1.1 (unreleased)

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 WCS Paper IV 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 (unreleased)

This is a bug fix release for the 3.0.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. 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 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 futher details and background. Some highlights are listed below:

- The Header class now fully implements the Python dict interface, and can be used interchangably 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 distict 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. See http://listmgr.cv.nrao.edu/pipermail/fitsbits/2002-April/001094.html. 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, recoverably) 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 unncessarily. (#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 psuedo-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 usecase 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 insignficant) 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 explictly 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 backards (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)
- Miscellanous 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 documention 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 compatability, 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 compatable 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 Paper IV 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 grammer 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 retrivial 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 retrivial 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):
File "<stdin>", line 1, in <module>
File "NP_pyfits.py", line 977, in __getitem__
  return self.ascard[key].value
File "NP_pyfits.py", line 1258, in __getitem__
  _key = self.index_of(key)
File "NP_pyfits.py", line 1403, in index_of
  raise KeyError, 'Keyword %s not found.' % `key`
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 HEIRARCH 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 the design of the open() function to remove the output_verify argument.
- Remove the groups argument in GroupsHDU's contructor.
- 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 instanciation 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 ths 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 copywright 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 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:
 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).

1.10.5 Reference/API

File Handling and Convenience Functions

`open()`

`astropy.io.fits.open`(*name*, *mode='readonly'*, *memmap=None*, *save_backup=False*, ***kwargs*)

Factory function to open a FITS file and return an `HDUList` object.

Parameters

name : file path, file object or file-like object

File to be opened.

mode : str

Open mode, 'readonly' (default), 'update', 'append', 'denywrite', or 'ostream'.

If `name` is a file object that is already opened, `mode` must match the mode the file was opened with, copyonwrite (rb), readonly (rb), update (rb+), append (ab+), ostream (w), denywrite (rb)).

memmap : bool

Is memory mapping to be used?

save_backup : bool

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.

kwargs : dict

optional keyword arguments, possible values are:

- **uint** : bool

  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.

  Note, for backward compatibility, the kwarg **uint16** may be used instead. The kwarg was renamed when support was added for integers of any size.

- **ignore_missing_end** : bool

  Do not issue an exception when opening a file that is missing an `END` card in the last header.

- **checksum** : bool, str

  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. Updates to a file that already has a checksum will preserve and update the existing checksums unless this argument is given a value of 'remove', in which case the CHECKSUM and DATASUM values are not checked, and are removed when saving changes to the file.

- **disable_image_compression** : bool

  If `True`, treates compressed image HDU's like normal binary table HDU's.

•**do_not_scale_image_data** : bool

If True, image data is not scaled using BSCALE/BZERO values when read.

•**scale_back** : bool

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.

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'*, *clobber=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 for append (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". See *Verification options* for more info.

clobber : bool (optional)

If True, and if filename already exists, it will overwrite the file. Default is False.

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+.

output : file (optional)

File-like object to output the HDU info to. Outputs to 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.

`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 :

Any additional keyword arguments to be passed to `astropy.io.fits.open`.

`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*, ***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.

view : ndarray (optional)

When given, the data will be turned wrapped in the given ndarray subclass by calling:

```
data.view(view)
```

kwargs :

Any additional keyword arguments to be passed to `astropy.io.fits.open`.

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 : string, integer, or float

`setval()`

`astropy.io.fits.setval`(*filename*, *keyword*, **args*, ***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

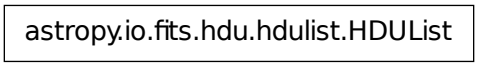
ext, extname, extver :

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.

`append(`*hdu*`)`
Append a new HDU to the `HDUList`.

Parameters
hdu : instance of _BaseHDU

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"`. See *Verification options* for more info.

verbose : bool

When `True`, print out verbose messages.

closed : bool

When `True`, close the underlying file 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, copyonwrite, 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(*args, **kwargs)`

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"`. See *Verification options* for more info.

verbose : bool

When `True`, print verbose messages

classmethod `fromfile(fileobj, mode='readonly', memmap=False, save_backup=False, **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 or tuple of (string, int)

The key identifying the HDU. 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.

Returns

index : int

The index of the HDU 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, 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 : _BaseHDU instance

The HDU object to insert

`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'*, *clobber=False*, *checksum=False*`)`

Write the `HDUList` to a new file.

Parameters

fileobj : file path, file object or file-like object

File to write to. If a file object, must be opened for append (ab+).

output_verify : str

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. See *Verification options* for more info.

clobber : bool

When `True`, overwrite the output file if exists.

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*, *uint=False*, *scale_back=False*`)`

Bases: `astropy.io.fits.hdu.image._ImageBaseHDU`

FITS primary HDU class.

add_checksum(*when=None*, *override_datasum=False*, *blocking='standard'*)
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

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

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*, *blocking='standard'*)
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

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

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.

filebytes()
Calculates and returns the number of bytes that this HDU will write to a file.

Parameters
None :

Returns
Number of bytes :

`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`.

Parameters

None :

Returns

dictionary 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
file-mode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
hdr-Loc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

classmethod `fromstring`(*data*, *fileobj=None*, *offset=0*, *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 contining the HDU's header and, optionally, its data. If `fileobj` is not specified, and the length of `data` extends beyond the header, then the trailing data is taken to be the HDU's data. If `fileobj` is specified then the trailing data is ignored.

fileobj : file (optional)

The file-like object that this HDU was read from.

offset : int (optional)

If `fileobj` is specified, the offset into the file-like object at which this HDU begins.

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 can't be found, so the entire data is just assumed to be the header.

kwargs : (optional)

May contain additional keyword arguments specific to an HDU type. Any unrecognized kwargs are simply ignored.

classmethod readfrom(*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with fitsopen() 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.

TODO: Write about parameters

If pos = None, it can be anywhere. 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/BZERO.

Call to this method will scale data and update the keywords of BSCALE and BZERO in _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

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.

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 PyFITS documentation for more details.

`shape`
Shape of the image array–should be equivalent to `self.data.shape`.

`size`
Size (in bytes) of the data portion of the HDU.

`update_ext_name`(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)
Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or 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.

Parameters

value : str

value to be used for the new extension name

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

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.

`update_ext_version`(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)
Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or 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.

Parameters

value : str

value to be used for the new extension version

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the Card after which the new card will be placed in the Header.

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.

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". See *Verification options* for more info.

verify_checksum(*blocking='standard'*)

Verify that the value in the CHECKSUM keyword matches the value calculated for the current HDU CHECKSUM.

blocking: str, optional

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no CHECKSUM keyword present

verify_datasum(*blocking='standard'*)

Verify that the value in the DATASUM keyword matches the value calculated for the DATASUM of the current HDU data.

blocking: str, optional

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no DATASUM keyword present

writeto(*name*, *output_verify='exception'*, *clobber=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 opened, must be opened for append ("ab+")).

output_verify : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". See *Verification options* for more info.

clobber : bool

Overwrite the output file if exists.

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*, *name=None*)

Bases: `astropy.io.fits.hdu.image.PrimaryHDU`, `astropy.io.fits.hdu.table._TableLikeHDU`

FITS Random Groups HDU class.

`add_checksum`(*when=None*, *override_datasum=False*, *blocking='standard'*)

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

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

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*, *blocking='standard'*)

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

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

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.

`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.

Parameters
None :

Returns
Number of bytes :

`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`.

Parameters
None :

Returns
dictionary 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*, *fileobj=None*, *offset=0*, *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 contining the HDU's header and, optionally, its data. If `fileobj` is not specified, and the length of `data` extends beyond the header, then the trailing data is taken to be the HDU's data. If `fileobj` is specified then the trailing data is ignored.

fileobj : file (optional)

The file-like object that this HDU was read from.

offset : int (optional)

If `fileobj` is specified, the offset into the file-like object at which this HDU begins.

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 can't be found, so the entire data is just assumed to be the header.

kwargs : (optional)

May contain additional keyword arguments specific to an HDU type. Any unrecognized kwargs are simply ignored.

`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 `fitsopen()` 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`.

TODO: Write about parameters

If `pos` = `None`, it can be anywhere. 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`/`BZERO`.

Call to this method will scale `data` and update the keywords of `BSCALE` and `BZERO` in `_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

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.

`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 PyFITS documentation for more details.

`shape`

Shape of the image array–should be equivalent to `self.data.shape`.

`size`

Returns the size (in bytes) of the HDU's data part.

`update_ext_name`(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)

Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or 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.

Parameters

value : str

value to be used for the new extension name

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

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.

update_ext_version(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)
Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or 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.

Parameters

value : str

value to be used for the new extension version

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

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.

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"`. See *Verification options* for more info.

verify_checksum(*blocking='standard'*)
Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU CHECKSUM.

blocking: str, optional
"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no `CHECKSUM` keyword present

verify_datasum(*blocking='standard'*)
Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

blocking: str, optional
"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no DATASUM keyword present

writeto(*name*, *output_verify='exception'*, *clobber=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 opened, must be opened for append ("ab+")).

output_verify : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". See *Verification options* for more info.

clobber : bool

Overwrite the output file if exists.

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

Bases: astropy.io.fits.fitsrec.FITS_rec

Random groups data object.

Allows structured access to FITS Group data in a manner analogous to tables.

par(*parname*)

Get the group parameter values.

Group

class astropy.io.fits.GroupData

Bases: astropy.io.fits.fitsrec.FITS_rec

Random groups data object.

Allows structured access to FITS Group data in a manner analogous to tables.

par(*parname*)

Get the group parameter values.

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()
```

close()
Close the physical FITS file.

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 IOError 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 IOError 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*=[ ], *txtfile*=*None*)
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.

`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.

`ascard`

Deprecated since version 3.1: Use the `cards` attribute instead. Returns a CardList object wrapping this Header; provided for backwards compatibility for the old API (where Headers had an underlying CardList).

`ascardlist`(**args*, ***kwargs*)

Deprecated since version 3.0: Use the `ascard` attribute instead. Returns a `CardList` object.

`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.

`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.

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 supercedes 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.

`fromTxtFile(*args, **kwargs)`

Deprecated since version 3.1: This is equivalent to `self.extend(Header.fromtextfile(fileobj), update=True, update_first=True)`. Note that there there is no direct equivalent to the `replace=True` option since `Header.fromtextfile()` returns a new `Header` instance. Input the header parameters from an ASCII file.

The input header cards will be used to update the current header. Therefore, when an input card key matches a card key that already exists in the header, that card will be updated in place. Any input cards that do not already exist in the header will be added. Cards will not be deleted from the header.

Parameters

fileobj : file path, file object or file-like object

Input header parameters file.

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.

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 `IOError` 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

String containing the entire 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).

Returns

header :

A new `Header` instance.

classmethod `fromtextfile(`*fileobj*, *endcard=False*`)`

Equivalent to `Header.fromfile(fileobj, sep='\n', endcard=False, padding=False)`.

`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.

`get_comment(`**args*, ***kwargs*`)`

Deprecated since version 3.1: Use `header['COMMENT']` instead. Get all comment cards as a list of string texts.

`get_history(`**args*, ***kwargs*`)`

Deprecated since version 3.1: Use `header['HISTORY']` instead. Get all history cards as a list of string texts.

`has_key(`**args*, ***kwargs*`)`

Deprecated since version 3.0: Use `key in header` syntax instead. Like `dict.has_key()`.

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(*idx*, *card*, *useblanks=True*)
Inserts a new keyword+value card into the Header at a given location, similar to list.insert().

Parameters
idx : int

The index into the the list of header keywords before which the new keyword should be inserted

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.

items()
Like dict.items().

iteritems()
Like dict.iteritems().

iterkeys()
Like dict.iterkeys()–iterating directly over the Header instance has the same behavior.

itervalues()
Like dict.itervalues().

keys()
Return a list of keywords in the header in the order they appear–like:meth:dict.keys but ordered.

pop(**args*)
Works like list.pop() if no arguments or an index argument are supplied; otherwise works like dict.pop().

popitem()

remove(*keyword*)
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
value : str

The keyword of which to remove the first instance in the header

rename_key(**args*, ***kwargs*)
Deprecated since version 3.1: Use `Header.rename_keyword()` instead.

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 PyFITS 3.1.

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)` respectfully.

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 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*`)`

`toTxtFile(`**args*, ***kwargs*`)`

Deprecated since version 3.1: Use `Header.totextfile()` instead. Output the header parameters to a file in ASCII format.

Parameters

fileobj : file path, file object or file-like object

Output header parameters file.

clobber : bool

When `True`, overwrite the output file if it exists.

`tofile(`*fileobj*, *sep=''*, *endcard=True*, *padding=True*, *clobber=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

clobber : bool (optional)

If `True`, overwrites the output file if it already exists

`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 : string

A string representing a FITS header.

`totextfile(`*fileobj*, *endcard=False*, *clobber=False*`)`

Equivalent to `Header.tofile(fileobj, sep='\n', endcard=False, padding=False, clobber=clobber)`.

`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 PyFITS versions prior to 3.1.0. However, support for the old API is also maintained for backwards compatibility. If update() is called with at least two positional arguments then it can be assumed that the old API is being used. Use of the old API should be considered **deprecated**. 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.
For reference, the old documentation for the old Header.update() is provided below:

Update one header card.

If the keyword already exists, it's value and/or 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.

Parameters

key : str

keyword

value : str

value to be used for updating

comment : str (optional)

to be used for updating, default=None.

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 specified.

after : str, int (optional)

name of the keyword, or index of the `Card` after which the new card will be placed.

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.

`values()`
Returns a list of the values of all cards in the header.

Cards

`Card`

class `astropy.io.fits.Card`(*keyword=None*, *value=None*, *comment=None*, ***kwargs*)
Bases: `astropy.io.fits.verify._Verify`

`ascardimage`(**args*, ***kwargs*)
Deprecated since version 3.1: Use the `image` attribute instead.

`cardimage`
Deprecated since version 3.1: Use the `image` attribute instead.

`comment`
Get the comment attribute from the card image if not already set.

`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).

`image`

`key`
Deprecated since version 3.1: Use the `keyword` attribute instead.

`keyword`
Returns the keyword name parsed from the card image.

`length` = **80**

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
key : or str

A keyword value or a `keyword.field-specifier` value

Returns
The converted string :

`rawvalue`

`run_option`(*option='warn'*, *err_text=''*, *fix_text='Fixed.'*, *fix=None*, *fixable=True*)
Execute the verification with selected option.

`value`

`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"`. See *Verification options* for more info.

Deprecated Interfaces

The following classes and functions are deprecated as of the PyFITS 3.1 header refactoring, though they are currently still available for backwards-compatibility.

class `astropy.io.fits.CardList`(*cards=[]*, *keylist=None*)
Bases: `list` Deprecated since version 3.1: `CardList` used to provide the list-like functionality for manipulating a header as a list of cards. This functionality is now subsumed into the `Header` class itself, so it is no longer necessary to create or use `CardList`s.

`append`(**args*, ***kwargs*)
Deprecated since version 3.1: Use `Header.append()` instead. Append a `Card` to the `CardList`.

Parameters
card : `Card` object

The `Card` to be appended.

useblanks : bool, optional

Use any *extra* blank cards?

If `useblanks` is `True`, and if there are blank cards directly before `END`, it will use this space first, instead of appending after these blank cards, so the total space will not increase. When `useblanks` is `False`, the card will be appended at the end, even if there are blank cards in front of `END`.

bottom : bool, optional

If `False` the card will be appended after the last non-commentary card. If `True` the card will be appended after the last non-blank card.

`copy`(**args*, ***kwargs*)
Deprecated since version 3.1: Use `Header.copy()` instead. Make a (deep)copy of the `CardList`.

`count`(**args*, ***kwargs*)
Deprecated since version 3.1: Use `Header.count()` instead.

`count_blanks`(**args*, ***kwargs*)
Deprecated since version 3.1: The count_blanks function is deprecated and may be removed in a future version. Returns how many blank cards are *directly* before the `END` card.

`extend(*args, **kwargs)`
Deprecated since version 3.1: Use `Header.extend()` instead.

`filterList(*args, **kwargs)`
Deprecated since version 3.0: Use `filter_list()` instead.

`filter_list(*args, **kwargs)`
Deprecated since version 3.1: Use `header[<wildcard_pattern>]` instead. Construct a `CardList` that contains references to all of the cards in this `CardList` that match the input key value including any special filter keys (`*`, ?, and ...).

Parameters
key : str

key value to filter the list with

Returns
cardlist : :

A `CardList` object containing references to all the requested cards.

`index(*args, **kwargs)`
Deprecated since version 3.1: Use `Header.index()` instead.

`index_of(*args, **kwargs)`
Deprecated since version 3.1: Use `Header.index()` instead. Get the index of a keyword in the `CardList`.

Parameters
key : str or int

The keyword name (a string) or the index (an integer).

backward : bool, (optional)

When `True`, search the index from the `END`, i.e., backward.

Returns
index : int

The index of the `Card` with the given keyword.

`insert(*args, **kwargs)`
Deprecated since version 3.1: Use `Header.insert()` instead. Insert a `Card` to the `CardList`.

Parameters
pos : int

The position (index, keyword name will not be allowed) to insert. The new card will be inserted before it.

card : `Card` object

The card to be inserted.

useblanks : bool, optional

If `useblanks` is `True`, and if there are blank cards directly before `END`, it will use this space first, instead of appending after these blank cards, so the total space will not increase. When `useblanks` is `False`, the card will be appended at the end, even if there are blank cards in front of `END`.

`keys(*args, **kwargs)`
Deprecated since version 3.1: Use `Header.keys()` instead. Return a list of all keywords from the `CardList`.

pop(**args*, ***kwargs*)
Deprecated since version 3.1: Use `Header.pop()` instead.

remove(**args*, ***kwargs*)
Deprecated since version 3.1: Use `Header.remove()` instead.

values(**args*, ***kwargs*)
Deprecated since version 3.1: Use `Header.values()` instead. Return a list of the values of all cards in the `CardList`.

For `RecordValuedKeywordCard` objects, the value returned is the floating point value, exclusive of the `field_specifier`.

astropy.io.fits.create_card(**args*, ***kwargs*)
Deprecated since version 3.1: Use `Card.__init__()` instead.

astropy.io.fits.create_card_from_string(**args*, ***kwargs*)
Deprecated since version 3.1: Use `Card.fromstring()` instead. 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).

astropy.io.fits.upper_key(**args*, ***kwargs*)
Deprecated since version 3.1: Use `Card.normalize_keyword()` instead. `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
key : or str

A keyword value or a `keyword.field-specifier` value

Returns
The converted string :

Tables

`BinTableHDU`

class astropy.io.fits.BinTableHDU(*data=None*, *header=None*, *name=None*)
Bases: `astropy.io.fits.hdu.table._TableBaseHDU`

Binary table HDU class.

add_checksum(*when=None*, *override_datasum=False*, *blocking='standard'*)
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

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

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*, *blocking='standard'*)

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

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

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 table HDU, both header and data are copied.

`dump`(*datafile=None*, *cdfile=None*, *hfile=None*, *clobber=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.

clobber : bool

Overwrite the output files if they exist.

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.

  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.

Parameters

None :

Returns

Number of bytes :

`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.

Parameters

None :

Returns

dictionary 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
file-mode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
hdr-Loc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

classmethod `fromstring`(*data*, *fileobj=None*, *offset=0*, *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 contining the HDU's header and, optionally, its data. If `fileobj` is not specified, and the length of `data` extends beyond the header, then the trailing data is taken to be the HDU's data. If `fileobj` is specified then the trailing data is ignored.

fileobj : file (optional)

The file-like object that this HDU was read from.

offset : int (optional)

If `fileobj` is specified, the offset into the file-like object at which this HDU begins.

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 can't be found, so the entire data is just assumed to be the header.

kwargs : (optional)

May contain additional keyword arguments specific to an HDU type. Any unrecognized kwargs are simply ignored.

`get_coldefs`(**args*, ***kwargs*)

Deprecated since version 3.0: Use the `columns` attribute instead. Returns the table's column definitions.

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

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 object

When the cdfile and hfile are missing, use this Header object in the creation of the new table and HDU. Otherwise this Header supercedes 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.

  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 `readfrom`(*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)
Read the HDU from a file. Normally an HDU should be opened with `fitsopen()` 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`.

TODO: Write about parameters

If `pos` = `None`, it can be anywhere. 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.

`size`
Size (in bytes) of the data portion of the HDU.

classmethod `tcreate`(**args*, ***kwargs*)
Deprecated since version 3.1: Use `load()` instead.

`tdump`(**args*, ***kwargs*)
Deprecated since version 3.1: Use `dump()` instead.

`update()`
Update header keywords to reflect recent changes of columns.

`update_ext_name`(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)
Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or 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.

Parameters
value : str

value to be used for the new extension name

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the Card before which the new card will be placed in the Header. The argument before takes precedence over after if both specified.

after : str or int, optional

name of the keyword, or index of the Card after which the new card will be placed in the Header.

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.

update_ext_version(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)

Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or 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.

Parameters

value : str

value to be used for the new extension version

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the Card before which the new card will be placed in the Header. The argument before takes precedence over after if both specified.

after : str or int, optional

name of the keyword, or index of the Card after which the new card will be placed in the Header.

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.

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". See *Verification options* for more info.

verify_checksum(*blocking='standard'*)

Verify that the value in the CHECKSUM keyword matches the value calculated for the current HDU CHECKSUM.

blocking: str, optional

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no CHECKSUM keyword present

verify_datasum(*blocking='standard'*)

Verify that the value in the DATASUM keyword matches the value calculated for the DATASUM of the current HDU data.

blocking: str, optional

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no DATASUM keyword present

writeto(*name*, *output_verify='exception'*, *clobber=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).

TableHDU

class astropy.io.fits.TableHDU(*data=None*, *header=None*, *name=None*)

Bases: astropy.io.fits.hdu.table._TableBaseHDU

FITS ASCII table extension HDU class.

add_checksum(*when=None*, *override_datasum=False*, *blocking='standard'*)

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

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

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*, *blocking='standard'*)
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

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

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 table HDU, both header and data are copied.

filebytes()
Calculates and returns the number of bytes that this HDU will write to a file.

Parameters
None :

Returns
Number of bytes :

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.

Parameters
None :

Returns
dictionary 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*, *fileobj=None*, *offset=0*, *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 contining the HDU's header and, optionally, its data. If fileobj is not specified, and the length of data extends beyond the header, then the trailing data is taken to be the HDU's data. If fileobj is specified then the trailing data is ignored.

fileobj : file (optional)

The file-like object that this HDU was read from.

offset : int (optional)

If fileobj is specified, the offset into the file-like object at which this HDU begins.

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 can't be found, so the entire data is just assumed to be the header.

kwargs : (optional)

May contain additional keyword arguments specific to an HDU type. Any unrecognized kwargs are simply ignored.

get_coldefs(**args*, ***kwargs*)

Deprecated since version 3.0: Use the columns attribute instead. Returns the table's column definitions.

classmethod readfrom(*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with fitsopen() 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.

TODO: Write about parameters

If pos = None, it can be anywhere. 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.

size
Size (in bytes) of the data portion of the HDU.

update()
Update header keywords to reflect recent changes of columns.

update_ext_name(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)
Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or 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.

Parameters

value : str

value to be used for the new extension name

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the Card before which the new card will be placed in the Header. The argument before takes precedence over after if both specified.

after : str or int, optional

name of the keyword, or index of the Card after which the new card will be placed in the Header.

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.

update_ext_version(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)
Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or 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.

Parameters

value : str

value to be used for the new extension version

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

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.

`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"`. See *Verification options* for more info.

`verify_checksum`(*blocking='standard'*)

Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU CHECKSUM.

blocking: str, optional

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no `CHECKSUM` keyword present

`verify_datasum`(*blocking='standard'*)

Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

blocking: str, optional

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no `DATASUM` keyword present

`writeto`(*name*, *output_verify='exception'*, *clobber=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).

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*)

Bases: object

Class which contains the definition of one column, e.g. ttype, tform, etc. and the array containing values for the column. Does not support theap yet.

copy()

Return a copy of this Column.

ColDefs

class astropy.io.fits.ColDefs(*input*, *tbtype='BinTableHDU'*)

Bases: object

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.

add_col(*column*)

Append one Column to the column definition.

change_attrib(*col_name*, *attrib*, *new_value*)

Change an attribute (in the commonName list) of a Column.

col_name

[str or int] The column name or index to change

attrib

[str] The attribute name

value

[object] The new value for the attribute

change_name(*col_name*, *new_name*)

Change a Column's name.

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.

col_name

[str or int] The column name or index

new_unit

[str] The new unit for the column

data

Deprecated since version 3.0: The data attribute is deprecated; use the ColDefs.columns attribute instead. What was originally self.columns is now self.data; this provides some backwards compatibility.

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 `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`
Bases: `numpy.core.records.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`.

`columns`
A user-visible accessor for the coldefs. See ticket #44.

`field(`*key*`)`
A view of a `Column`'s data as an array.

`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.

`field(`*field*`)`
Get the field data of the record.

`setfield(`*field*, *value*`)`
Set the field data of the record.

Table Functions

new_table()

astropy.io.fits.new_table(*input*, *header=None*, *nrows=0*, *fill=False*, *tbtype='BinTableHDU'*)

Create a new table from the input column definitions.

Warning: Creating a new table using this method creates an in-memory *copy* of all the column arrays in the input. This is because if they are separate arrays they must be combined into a single contiguous array.

If the column data is already in a single contiguous array (such as an existing record array) it may be better to create a BinTableHDU instance directly. See the PyFITS documentation for more details.

Parameters

input : sequence of Column or ColDefs objects

The data to create a table from.

header : Header instance

Header to be used to populate the non-required keywords.

nrows : int

Number of rows in the new table.

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.

tbtype : str

Table type to be created ("BinTableHDU" or "TableHDU").

tabledump()

astropy.io.fits.tabledump(*filename*, *datafile=None*, *cdfile=None*, *hfile=None*, *ext=1*, *clobber=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.

clobber : bool

Overwrite the output files if they exist.

Notes

The primary use for the `tabledump` function is to allow editing in a standard text editor of the table data and parameters. The `tcreate` 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.

  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 multi-dimensional 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.

  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 multi-dimensional 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.

Images

ImageHDU

class astropy.io.fits.ImageHDU(*data=None*, *header=None*, *name=None*, *do_not_scale_image_data=False*, *uint=False*, *scale_back=False*)

Bases: astropy.io.fits.hdu.image._ImageBaseHDU, astropy.io.fits.hdu.base.ExtensionHDU

FITS image extension HDU class.

add_checksum(*when=None*, *override_datasum=False*, *blocking='standard'*)

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

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

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*, *blocking='standard'*)

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

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

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.

`filebytes()`

Calculates and returns the number of bytes that this HDU will write to a file.

Parameters

None :

Returns

Number of bytes :

`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`.

Parameters

None :

Returns

dictionary 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
file-mode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
hdr-Loc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

classmethod `fromstring`(*data*, *fileobj=None*, *offset=0*, *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 contining the HDU's header and, optionally, its data. If `fileobj` is not specified, and the length of `data` extends beyond the header, then the trailing data is taken to be the HDU's data. If `fileobj` is specified then the trailing data is ignored.

fileobj : file (optional)

The file-like object that this HDU was read from.

offset : int (optional)

If `fileobj` is specified, the offset into the file-like object at which this HDU begins.

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 can't be found, so the entire data is just assumed to be the header.

kwargs : (optional)

May contain additional keyword arguments specific to an HDU type. Any unrecognized kwargs are simply ignored.

classmethod `readfrom`(*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with `fitsopen()` 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.

TODO: Write about parameters

If pos = None, it can be anywhere. 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/BZERO.

Call to this method will scale data and update the keywords of BSCALE and BZERO in _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

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.

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 PyFITS documentation for more details.

shape
Shape of the image array–should be equivalent to self.data.shape.

size
Size (in bytes) of the data portion of the HDU.

update_ext_name(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)
Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or 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.

Parameters

value : str

value to be used for the new extension name

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

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.

`update_ext_version`(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)

Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or 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.

Parameters

value : str

value to be used for the new extension version

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

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.

`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"`. See *Verification options* for more info.

`verify_checksum`(*blocking='standard'*)
Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU CHECKSUM.

blocking: str, optional
"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns
valid : int

- 0 - failure
- 1 - success
- 2 - no `CHECKSUM` keyword present

`verify_datasum`(*blocking='standard'*)
Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

blocking: str, optional
"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns
valid : int

- 0 - failure
- 1 - success
- 2 - no `DATASUM` keyword present

`writeto`(*name*, *output_verify='exception'*, *clobber=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).

`CompImageHDU`

class `astropy.io.fits.CompImageHDU`(*data=None*, *header=None*, *name=None*, *compressionType='RICE_1'*, *tileSize=None*, *hcompScale=0*, *hcompSmooth=0*, *quantizeLevel=16.0*, *do_not_scale_image_data=False*, *uint=False*, *scale_back=False*, ***kwargs*)
Bases: `astropy.io.fits.hdu.table.BinTableHDU`

Compressed Image HDU class.

`add_checksum`(*when=None*, *override_datasum=False*, *blocking='standard'*)
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

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

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*, *blocking='standard'*)

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

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

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 table HDU, both header and data are copied.

dump(*datafile=None*, *cdfile=None*, *hfile=None*, *clobber=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.

clobber : bool

Overwrite the output files if they exist.

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.

  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.

Parameters
None :

Returns
Number of bytes :

`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`.

Parameters
None :

Returns
dictionary 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
file-mode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
hdr-Loc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

classmethod fromstring(*data*, *fileobj=None*, *offset=0*, *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 contining the HDU's header and, optionally, its data. If fileobj is not specified, and the length of data extends beyond the header, then the trailing data is taken to be the HDU's data. If fileobj is specified then the trailing data is ignored.

fileobj : file (optional)

The file-like object that this HDU was read from.

offset : int (optional)

If fileobj is specified, the offset into the file-like object at which this HDU begins.

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 can't be found, so the entire data is just assumed to be the header.

kwargs : (optional)

May contain additional keyword arguments specific to an HDU type. Any unrecognized kwargs are simply ignored.

get_coldefs(**args*, ***kwargs*)

Deprecated since version 3.0: Use the columns attribute instead. Returns the table's column definitions.

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

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 object

When the cdfile and hfile are missing, use this Header object in the creation of the new table and HDU. Otherwise this Header supercedes 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.

  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 readfrom(*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with fitsopen() 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.

TODO: Write about parameters

If pos = None, it can be anywhere. 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.

`shape`
Shape of the image array–should be equivalent to `self.data.shape`.

`size`
Size (in bytes) of the data portion of the HDU.

classmethod `tcreate(*args, **kwargs)`
Deprecated since version 3.1: Use `load()` instead.

`tdump(*args, **kwargs)`
Deprecated since version 3.1: Use `dump()` instead.

`update()`
Update header keywords to reflect recent changes of columns.

`updateCompressedData()`
Compress the image data so that it may be written to a file.

`updateHeader()`
Update the table header cards to match the compressed data.

`updateHeaderData(image_header, name=None, compressionType=None, tileSize=None, hcompScale=None, hcompSmooth=None, quantizeLevel=None)`
Update the table header (`_header`) to the compressed image format and to match the input data (if any). Create the image header (`_image_header`) from the input image header (if any) and ensure it matches the input data. Create the initially-empty table data array to hold the compressed data.

This method is mainly called internally, but a user may wish to call this method after assigning new data to the `CompImageHDU` object that is of a different type.

Parameters

image_header : Header instance

header to be associated with the image

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

compressionType : str, optional

compression algorithm 'RICE_1', 'PLIO_1', 'GZIP_1', 'HCOMPRESS_1'; if this value is `None`, use value already in the header; if no value already in the header, use 'RICE_1'

tileSize : sequence of int, optional

compression tile sizes as a list; if this value is `None`, use value already in the header; if no value already in the header, treat each row of image as a tile

hcompScale : float, optional

HCOMPRESS scale parameter; if this value is `None`, use the value already in the header; if no value already in the header, use 1

hcompSmooth : float, optional

HCOMPRESS smooth parameter; if this value is `None`, use the value already in the header; if no value already in the header, use 0

quantizeLevel : float, optional

floating point quantization level; if this value is None, use the value already in the header; if no value already in header, use 16

update_ext_name(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)
Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or 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.

Parameters

value : str

value to be used for the new extension name

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

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.

update_ext_version(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)
Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or 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.

Parameters

value : str

value to be used for the new extension version

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

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.

`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"`. See *Verification options* for more info.

`verify_checksum(`*blocking='standard'*`)`
Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU CHECKSUM.

blocking: str, optional
"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns
valid : int

- 0 - failure
- 1 - success
- 2 - no `CHECKSUM` keyword present

`verify_datasum(`*blocking='standard'*`)`
Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

blocking: str, optional
"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns
valid : int

- 0 - failure
- 1 - success
- 2 - no `DATASUM` keyword present

`writeto(`*name*, *output_verify='exception'*, *clobber=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).

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_keywords*=[], *ignore_comments*=[], *ignore_fields*=[], *numdiffs=10*, *tolerance=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.

classmethod fromdiff(*other*, *a*, *b*)

Returns a new Diff object of a specfic subclass from an existing diff object, passing on the values for any arguments they share in common (such as ignore_keywords).

For example:

```
>>> fd = FITSDiff('a.fits', 'b.fits', ignore_keywords=['*'])
>>> hd = HeaderDiff.fromdiff(fd, header_a, header_b)
>>> hd.ignore_keywords
['*']
```

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*)

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 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).

indent : int

The number of 4 space tabs to indent the report.

Returns

report : str or None

HDUDiff

class astropy.io.fits.HDUDiff(*a*, *b*, *ignore_keywords*=[], *ignore_comments*=[], *ignore_fields*=[], *numdiffs=10*, *tolerance=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_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).

classmethod fromdiff(*other*, *a*, *b*)

Returns a new Diff object of a specfic subclass from an existing diff object, passing on the values for any arguments they share in common (such as ignore_keywords).

For example:

```
>>> fd = FITSDiff('a.fits', 'b.fits', ignore_keywords=['*'])
>>> hd = HeaderDiff.fromdiff(fd, header_a, header_b)
>>> hd.ignore_keywords
['*']
```

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*)

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 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).

indent : int

The number of 4 space tabs to indent the report.

Returns

report : str or None

HeaderDiff

class astropy.io.fits.HeaderDiff(*a*, *b*, *ignore_keywords=[]*, *ignore_comments=[]*, *tolerance=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.

classmethod `fromdiff`(*other*, *a*, *b*)

Returns a new Diff object of a specfic subclass from an existing diff object, passing on the values for any arguments they share in common (such as ignore_keywords).

For example:

```
>>> fd = FITSDiff('a.fits', 'b.fits', ignore_keywords=['*'])
>>> hd = HeaderDiff.fromdiff(fd, header_a, header_b)
>>> hd.ignore_keywords
['*']
```

`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*)

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 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).

indent : int

The number of 4 space tabs to indent the report.

Returns
report : str or None

ImageDataDiff

class astropy.io.fits.ImageDataDiff(*a*, *b*, *numdiffs=10*, *tolerance=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.

classmethod fromdiff(*other*, *a*, *b*)
Returns a new Diff object of a specfic subclass from an existing diff object, passing on the values for any arguments they share in common (such as ignore_keywords).

For example:

```
>>> fd = FITSDiff('a.fits', 'b.fits', ignore_keywords=['*'])
>>> hd = HeaderDiff.fromdiff(fd, header_a, header_b)
>>> hd.ignore_keywords
['*']
```

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*)
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 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).

indent : int

The number of 4 space tabs to indent the report.

Returns

report : str or None

RawDataDiff

class astropy.io.fits.RawDataDiff(*a*, *b*, *numdiffs=10*)

Bases: astropy.io.fits.diff.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.

classmethod fromdiff(*other*, *a*, *b*)

Returns a new Diff object of a specfic subclass from an existing diff object, passing on the values for any arguments they share in common (such as ignore_keywords).

For example:

```
>>> fd = FITSDiff('a.fits', 'b.fits', ignore_keywords=['*'])
>>> hd = HeaderDiff.fromdiff(fd, header_a, header_b)
>>> hd.ignore_keywords
['*']
```

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*)

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 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).

indent : int

The number of 4 space tabs to indent the report.

Returns

report : str or None

TableDataDiff

class astropy.io.fits.TableDataDiff(*a*, *b*, *ignore_fields=[]*, *numdiffs=10*, *tolerance=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 secondard 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.

classmethod fromdiff(*other*, *a*, *b*)
Returns a new Diff object of a specfic subclass from an existing diff object, passing on the values for any arguments they share in common (such as ignore_keywords).

For example:

```
>>> fd = FITSDiff('a.fits', 'b.fits', ignore_keywords=['*'])
>>> hd = HeaderDiff.fromdiff(fd, header_a, header_b)
>>> hd.ignore_keywords
['*']
```

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*)
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 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).

indent : int

The number of 4 space tabs to indent the report.

Returns

report : str or None

Verification options

There are 5 options for the `output_verify` argument of the following methods of `HDUList`: `close()`, `writeto()`, and `flush()`, or the `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 wil 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 violation: 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's 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 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.

1.11 ASCII Tables (`astropy.io.ascii`)

1.11.1 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 ease of use.

The following formats are supported:

- `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
- `Daophot`: table from the IRAF DAOphot package
- `FixedWidth`: table with fixed-width columns (see also *Fixed-width Gallery*)
- `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
- `SExtractor`: SExtractor format table
- `Tab`: tab-separated values

The `astropy.io.ascii` package is built on a modular and extensible class structure with independent *Base class elements* so that new formats can be easily accomodated.

Note: It is also possible to use the functionality from `astropy.io.ascii` through a higher-level interface in the `astropy.table` package. See *Reading and writing Table objects* for more details.

1.11.2 Getting Started

Reading Tables

The majority of commonly encountered ASCII tables can be easily 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. By default `read()` will try to guess the table format by trying all the supported formats. If this does not work (for unusually formatted tables) then one needs give astropy.io.ascii additional hints about the format, for example:

```
>>> 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
```

Writing Tables

The `write()` function provides a way to write a data table as a formatted ASCII table. For example the following writes a table as a simple space-delimited file:

```
>>> x = np.array([1, 2, 3])
>>> y = x ** 2
>>> data = Table()
>>> ascii.write([x, y], names=['x', 'y'], 'values.dat')
```

The `values.dat` file will then contain:

```
x y
1 1
2 4
3 9
```

All 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:

```
>>> ascii.write(data, sys.stdout, Writer=ascii.Latex)
\begin{table}
\begin{tabular}{cc}
```

```
x & y \\
1 & 1 \\
2 & 4 \\
3 & 9 \\
\end{tabular}
\end{table}
```

1.11.3 Using `io.ascii`

The details of using `astropy.io.ascii` are provided in the following sections:

Reading tables

The majority of commonly encountered ASCII tables can be easily read with the `read()` function:

```
>>> from astropy.io import ascii
>>> data = ascii.read(table)
```

where `table` is the name of a file, a string representation of a table, or a list of table lines. By default `read()` will try to guess the table format by trying all the supported formats. If this does not work (for unusually formatted tables) then one needs give `astropy.io.ascii` additional hints about the format, for example:

```
>>> data = astropy.io.ascii.read('t/nls1_stackinfo.dbout', data_start=2, delimiter='|')
>>> data = astropy.io.ascii.read('t/simple.txt', quotechar="'")
>>> data = astropy.io.ascii.read('t/simple4.txt', Reader=ascii.NoHeader, delimiter='|')
```

The `read()` function accepts a number of parameters that specify the detailed table format. Different Reader classes can define different defaults, so the descriptions below sometimes mention "typical" default values. This refers to the `Basic` reader and other similar Reader classes.

Parameters for `read()`

table
: [input table] There are four ways to specify the table to be read:

  - Name of 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.

Reader
: [Reader class (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 a Reader class. For basic usage this means one of the built-in *Extension Reader classes*.

guess: try to guess table format (default=True)
: If set to True then `read()` will try to guess the table format by cycling through a number of possible table format permuations and attemping 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` Reader 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 not counting comment lines] This specifies in the line index where the header line will be found. Comment lines are not included in this count and the counting starts from 0 (first non-comment line has index=0). If set to None this indicates that there is no header line and the column names will be auto-generated. The default is dependent on the Reader.

data_start: line index for the start of data not counting comment lines
This specifies in the line index where the data lines begin where the counting starts from 0 and does not include comment lines. The default is dependent on the Reader.

data_end: line index for the end of data (can be negative to count from end)
If this is not None then it allows for excluding lines at the end that are not valid data lines. A negative value means to count from the end, so -1 would exclude the last line, -2 the last two lines, and so on.

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 exlude 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: fill value specifier of lists
This can be used to fill missing values in the table or replace strings with special meaning. See the Replace bad or missing values section for more information and examples.

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.

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. The other option is `NumpyOutputter` which returns a `numpy` record array. If `fill_values` are specified then `NumpyOutputter` is used and a masked array is returned.

Inputter: Inputter class
This is generally not specified.

data_Splitter: Splitter class to split data columns

header_Splitter: Splitter class to split header columns

Replace bad or missing values

astropy.io.ascii can replace string values in the input data before they are converted. The most common use case is probably a table which contains string values that are not a valid representation of a number, e.g. "..." for a missing value or "". If astropy.io.ascii cannot convert all elements in a column to a numeric type, it will format the column as strings. To avoid this, fill_values can be used at the string level to fill missing values with the following syntax, which replaces <old> with <new> before the type conversion is done:

```
fill_values = <fill_spec> | [<fill_spec1>, <fill_spec2>, ...]
<fill_spec> = (<old>, <new>, <optional col name 1>, <optional col name 2>, ...)
```

Within the <fill_spec> tuple the <old> and <new> values must be strings. These two values are then followed by zero or more column names. If column names are included the replacement is limited to those columns listed. If no columns are specified then the replacement is done in every column, subject to filtering by fill_include_names and fill_exclude_names (see below).

The fill_values parameter in read() takes a single <fill_spec> or a list of <fill_spec> tuples. If several <fill_spec> apply to a single occurence of <old> then the first one determines the <new> value. For instance the following will replace an empty data value in the x or y columns with "1e38" while empty values in any other column will get "-999":

```
>>> ascii.read(table, fill_values=[('', '1e38', 'x', 'y'), ('', '-999')])
```

The following shows an example where string information needs to be exchanged before the conversion to float values happens. Here no_rain and no_snow is replaced by 0.0:

```
>>> table = ['day  rain     snow',    # column names
             #---  -------  --------
             'Mon  3.2      no_snow',
             'Tue  no_rain  1.1',
             'Wed  0.3      no_snow']
>>> print ascii.read(table, fill_values=[('no_rain', '0.0'), ('no_snow', '0.0')])
[('Mon', 3.2, --) ('Tue', --, 1.1) ('Wed', 0.3, --)]
```

Sometimes these rules apply only to specific columns in the table. Columns can be selected with fill_include_names or excluded with fill_exclude_names. Also, column names can be given directly with fill_values:

```
>>> asciidata = ['text,no1,no2', 'text1,1,1.',',2,']
>>> print ascii.read(asciidata, fill_values = ('', 'nan','no1','no2'), delimiter = ',')
[('text1', 1, 1.0) ('', 2, --)]
```

Here, the empty value " in column no2 is replaced by nan, but the text column remains unaltered.

When fill_values is specified then read() returns a NumPy masked array instead of the default Table object. See the description of the NumpyOutputter class for information on disabling masked arrays.

Guess table format

If the guess parameter in `read()` is set to True (which is the default) then `read()` will try to guess the table format by cycling through a number of possible table format permutations and attemping 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.

The order of guessing is shown by this Python code:

```
for Reader in (Rdb, Tab, Cds, Daophot, SExtractor, Ipac):
    read(Reader=Reader)
for Reader in (CommentedHeader, Basic, 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 one must explicitly specify the reader class with the `Reader` keyword.

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 that were supplied to the read() function. Any guesses that would conflict are skipped. For example the call:

```
>>> data = astropy.io.ascii.read(table, Reader=NoHeader, quotechar="'")
```

would only try the four delimiter possibilities, skipping all the conflicting Reader and quotechar combinations.

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
```

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 2-element tuple `(converter_func, converter_type)` as described in the previous section. The type provided to `convert_numpy()` must be a valid numpy type, for example `numpy.int`, `numpy.uint`, `numpy.int8`, `numpy.int64`, `numpy.float`, `numpy.float64`, `numpy.str`.

The default converters for each column can be overridden with the `converters` keyword:

```
>>> converters = {'col1': [astropy.io.ascii.convert_numpy(numpy.uint)],
                  'col2': [astropy.io.ascii.convert_numpy(numpy.float32)]}
>>> ascii.read('file.dat', converters=converters)
```

Advanced customization

Here we provide a few examples that demonstrate how to extend the base functionality to handle special cases. To go beyond these simple examples the best reference is to read the code for the existing *Extension Reader classes*.

Define a custom reader functionally

```
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)
```

Define custom readers by class inheritance

```
# Note: Tab and Rdb are already included in astropy.io.ascii for convenience.
class Tab(astropy.io.ascii.Basic):
    def __init__(self):
        astropy.io.ascii.Basic.__init__(self)
        self.header.splitter.delimiter = '\t'
        self.data.splitter.delimiter = '\t'
        # Don't strip line whitespace since that includes tabs
        self.header.splitter.process_line = None
        self.data.splitter.process_line = None
        # Don't strip data value spaces since that is significant in TSV tables
        self.data.splitter.process_val = None
        self.data.splitter.skipinitialspace = False

class Rdb(astropy.io.ascii.Tab):
    def __init__(self):
        astropy.io.ascii.Tab.__init__(self)
        self.data.start_line = 2
```

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
```

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. For example:

```
>>> from astropy.io import ascii
>>> x = np.array([1, 2, 3])
>>> y = x ** 2
>>> ascii.write([x, y], names=['x', 'y'], 'values.dat')
```

The `values.dat` file will then contain:

```
x y
1 1
2 4
3 9
```

All of the input Reader table 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:

```
>>> ascii.write(data, sys.stdout, Writer=ascii.Latex)
\begin{table}
\begin{tabular}{cc}
x & y \\
1 & 1 \\
2 & 4 \\
3 & 9 \\
\end{tabular}
\end{table}
```

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). The sections below show a few examples.

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.

```
>>> 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, sys.stdout)
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, sys.stdout)
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 = astropy.io.ascii.read('t/daophot.dat', Reader=astropy.io.ascii.Daophot)
>>> astropy.io.ascii.write(data, 'space_delimited_table.dat')
```

List of lists A list of Python lists (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, sys.stdout)
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, sys.stdout, names=['x', 'y', 'z'], exclude_names=['y'])
x z
1 hello
2 world
3 !!!
```

Dict of lists 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.

```
>>> data = {'x': [1, 2, 3],
            'y': [4, 5.2, 6.1],
            'z': ['hello', 'world', '!!!']}
>>> ascii.write(data, sys.stdout, 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. Different Reader classes can define different defaults, so the descriptions below sometimes mention "typical" default values. This refers to the `Basic` reader and other similar Reader classes.

Some Reader classes, e.g. `Latex` or `AASTex` accept aditional 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*).

Writer
: [Writer class (default= `Basic`)] This specifies the top-level format of the ASCII table to be written, 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*. Note: Reader classes and Writer classes are synonymous, in other words Reader classes can also write, but for historical reasons they are called Reader classes.

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 a comment line in table] For the `Basic` Reader this defaults to "#".

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 exlude 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: fill value specifier of lists
This can be used to fill missing values in the table or replace values with special meaning. The syntax is the same as used on input. See the *Replace bad or missing values* section for more information on the syntax. When writing a table, all values are converted to strings, before any value is replaced. Thus, you need to provide the string representation (stripped of whitespace) for each value. Example:

```
astropy.io.ascii.write(table, sys.stdout, fill_values = [('nan', 'no data'),
                                                         ('-999.0', 'no data')])
```

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.

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 signicant 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, whether the delimiter appears on either end ("bookends"), and padding around the delimiter.

Details are available in the class API documentation, but the easiest way to understand all 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, Reader=ascii.FixedWidth)
rec.array([(1.2, '"hello"'), (2.4, "'s worlds")],
      dtype=[('Col1', '<f8'), ('Col2', '|S9')])
```

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, Reader=ascii.FixedWidth, names=('name1', 'name2'))
rec.array([(1.2, '"hello"'), (2.4, "'s worlds")],
      dtype=[('name1', '<f8'), ('name2', '|S9')])
```

Weird input table with data values chopped by col extent

```
>>> table = """
...   Col1  |  Col2 |
...   1.2       "hello"
...   2.4   sdf's worlds
... """
>>> ascii.read(table, Reader=ascii.FixedWidth)
rec.array([(1.2, '"hel'), (2.4, "df's wo")],
      dtype=[('Col1', '<f8'), ('Col2', '|S7')])
```

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, Reader=ascii.FixedWidth)
rec.array([('John', '555-1234', '192.168.1.10'),
       ('Mary', '555-2134', '192.168.1.12'),
       ('Bob', '555-4527', '192.168.1.9')],
      dtype=[('Name', '|S4'), ('Phone', '|S8'), ('TCP', '|S12')])
```

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, Reader=ascii.FixedWidth, delimiter=' ')
rec.array([('John', '555-1234', '192.168.1.10'),
       ('Mary', '555-2134', '192.168.1.12'),
       ('Bob', '555-4527', '192.168.1.9')],
      dtype=[('Name', '|S4'), ('--Phone-', '|S8'), ('----TCP-----', '|S12')])
```

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, Reader=ascii.FixedWidth,
...                header_start=None, data_start=0)
rec.array([('John', '555-1234', '192.168.1.10'),
       ('Mary', '555-2134', '192.168.1.12'),
       ('Bob', '555-4527', '192.168.1.9')],
      dtype=[('col1', '|S4'), ('col2', '|S8'), ('col3', '|S12')])
```

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, Reader=ascii.FixedWidth,
...                header_start=None, data_start=0,
...                names=('Name', 'Phone', 'TCP'))
rec.array([('John', '555-1234', '192.168.1.10')],
      dtype=[('Name', '|S4'), ('Phone', '|S8'), ('TCP', '|S12')])
```

FixedWidthNoHeader Table with no header row and auto-column naming. Use the FixedWidthNoHeader convenience class.

```
>>> 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, Reader=ascii.FixedWidthNoHeader)
rec.array([('John', '555-1234', '192.168.1.10'),
       ('Mary', '555-2134', '192.168.1.12'),
       ('Bob', '555-4527', '192.168.1.9')],
      dtype=[('col1', '|S4'), ('col2', '|S8'), ('col3', '|S12')])
```

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 0 to 5 will select the first 6 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, Reader=ascii.FixedWidthNoHeader,
...                names=('Name', 'Phone', 'TCP'),
...                col_starts=(0, 9, 18),
```

```
...                 col_ends=(5, 17, 28),
...                 )
rec.array([('John', '555- 1234', '192.168.1.'),
       ('Mary', '555- 2134', '192.168.1.'),
       ('Bob', '555- 4527', '192.168.1')],
      dtype=[('Name', '|S4'), ('Phone', '|S9'), ('TCP', '|S10')])
```

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, Reader=ascii.FixedWidthTwoLine)
rec.array([(1.2, '"hello"'), (2.4, "'s worlds")],
      dtype=[('Col1', '<f8'), ('Col2', '|S9')])
```

Restructured text table

```
>>> table = """
... ======= ===========
...   Col1    Col2
... ======= ===========
...   1.2   "hello"
...   2.4   's worlds
... ======= ===========
... """
>>> ascii.read(table, Reader=ascii.FixedWidthTwoLine,
...                 header_start=1, position_line=2, data_end=-1)
rec.array([(1.2, '"hello"'), (2.4, "'s worlds")],
      dtype=[('Col1', '<f8'), ('Col2', '|S9')])
```

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, Reader=ascii.FixedWidthTwoLine, delimiter='+',
...                 header_start=1, position_line=0, data_start=3, data_end=-1)
rec.array([(1.2, '"hello"'), (2.4, "'s worlds")],
      dtype=[('Col1', '<f8'), ('Col2', '|S9')])
```

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, Reader=ascii.FixedWidth)
```

Write a table as a normal fixed width table.

```
>>> ascii.write(dat, Writer=ascii.FixedWidth)
| 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, Writer=ascii.FixedWidth, 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, Writer=ascii.FixedWidth, 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, Writer=ascii.FixedWidth, 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, Writer=ascii.FixedWidth,
...                  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, Writer=ascii.FixedWidthNoHeader)
| 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, Writer=ascii.FixedWidthNoHeader, 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, Writer=ascii.FixedWidthNoHeader, 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, Writer=ascii.FixedWidthNoHeader, 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, Writer=ascii.FixedWidthTwoLine)
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, Writer=ascii.FixedWidthTwoLine,
...                  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, Writer=ascii.FixedWidthTwoLine, bookend=True, delimiter='|')
|Col1|     Col2|Col3|Col4|
|----|---------|----|----|
| 1.2|  "hello"|   1|   a|
| 2.4|'s worlds|   2|   2|
```

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`: Get the lines from the table input.
- `Splitter`: Split the lines into string column values.

- `Header`: Initialize output columns based on the table header or user input.
- `Data`: Populate column data from the table.
- `Outputter`: Convert 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 tweakable parameters is modularized into managable groups. Where it makes sense these attributes are actually functions that make it easy to handle special cases.

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
- `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
- `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
- `SExtractor`: SExtractor format table
- `Tab`: tab-separated values

1.11.4 Reference/API

astropy.io.ascii Module

An extensible ASCII table reader and writer.

Functions

`convert_numpy`(numpy_type)	Return a tuple `(converter_func, converter_type)`.
`get_reader`([Reader, Inputter, Outputter])	Initialize a table reader allowing for common customizations.
`get_writer`([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, Writer])	Write the input `table` to `filename`.

convert_numpy

astropy.io.ascii.core.convert_numpy(*numpy_type*)

Return a tuple (converter_func, converter_type). The converter function converts a list into a numpy array of the given numpy_type. This type must be a valid numpy type, e.g. numpy.int, numpy.uint, numpy.int8, numpy.int64, numpy.float, numpy.float64, numpy.str. The converter type is used to track the generic data type (int, float, str) that is produced by the converter function.

get_reader

astropy.io.ascii.ui.get_reader(*Reader=None*, *Inputter=None*, *Outputter=None*, ***kwargs*)

Initialize a table reader allowing for common customizations. Most of the default behavior for various parameters is determined by the Reader class.

Parameters

- **Reader** – Reader class (default= Basic)
- **Inputter** – Inputter class
- **Outputter** – Outputter class
- **delimiter** – column delimiter string
- **comment** – regular expression defining a comment line in table
- **quotechar** – one-character string to quote fields containing special characters
- **header_start** – line index for the header line not counting comment lines
- **data_start** – line index for the start of data not counting comment lines
- **data_end** – line index for the end of data (can be negative to count from end)
- **converters** – dict of converters
- **data_Splitter** – Splitter class to split data columns
- **header_Splitter** – Splitter class to split header columns
- **names** – list of names corresponding to each data column
- **include_names** – list of names to include in output (default=None selects all names)
- **exclude_names** – list of names to exlude from output (applied after include_names)
- **fill_values** – specification of fill values for bad or missing table values
- **fill_include_names** – list of names to include in fill_values (default=None selects all names)
- **fill_exclude_names** – list of names to exlude from fill_values (applied after fill_include_names)

get_writer

astropy.io.ascii.ui.get_writer(*Writer=None*, ***kwargs*)

Initialize a table writer allowing for common customizations. Most of the default behavior for various parameters is determined by the Writer class.

Parameters

- **Writer** – Writer class (default=‘‘ascii.Basic‘‘)
- **delimiter** – column delimiter string

- **write_comment** – string defining a comment line in table
- **quotechar** – one-character string to quote fields containing special characters
- **formats** – dict of format specifiers or formatting functions
- **strip_whitespace** – strip surrounding whitespace from column values (default=True)
- **names** – list of names corresponding to each data column
- **include_names** – list of names to include in output (default=None selects all names)
- **exclude_names** – list of names to exlude from output (applied after `include_names`)

read

`astropy.io.ascii.ui.read`(*table*, *guess=None*, ***kwargs*)

Read the input `table` and return the table. Most of the default behavior for various parameters is determined by the Reader class.

Parameters

- **table** – input table (file name, list of strings, or single newline-separated string)
- **guess** – try to guess the table format (default=True)
- **Reader** – Reader class (default=‘‘ascii.Basic‘‘)
- **Inputter** – Inputter class
- **Outputter** – Outputter class
- **delimiter** – column delimiter string
- **comment** – regular expression defining a comment line in table
- **quotechar** – one-character string to quote fields containing special characters
- **header_start** – line index for the header line not counting comment lines
- **data_start** – line index for the start of data not counting comment lines
- **data_end** – line index for the end of data (can be negative to count from end)
- **converters** – dict of converters
- **data_Splitter** – Splitter class to split data columns
- **header_Splitter** – Splitter class to split header columns
- **names** – list of names corresponding to each data column
- **include_names** – list of names to include in output (default=None selects all names)
- **exclude_names** – list of names to exlude from output (applied after `include_names`)
- **fill_values** – specification of fill values for bad or missing table values
- **fill_include_names** – list of names to include in fill_values (default=None selects all names)
- **fill_exclude_names** – list of names to exlude from fill_values (applied after `fill_include_names`)

set_guess
astropy.io.ascii.ui.set_guess(*guess*)
Set the default value of the guess parameter for read()

Parameters
guess – New default guess value (TruelFalse)

write
astropy.io.ascii.ui.write(*table*, *output=<open file '<stdout>', mode 'w' at 0x7f23551061e0>*, *Writer=None*, ***kwargs*)
Write the input table to filename. Most of the default behavior for various parameters is determined by the Writer class.

Parameters

- **table** – input table (Reader object, NumPy struct array, list of lists, etc)
- **output** – output [filename, file-like object] (default = sys.stdout)
- **Writer** – Writer class (default=''ascii.Basic'')
- **delimiter** – column delimiter string
- **write_comment** – string defining a comment line in table
- **quotechar** – one-character string to quote fields containing special characters
- **formats** – dict of format specifiers or formatting functions
- **strip_whitespace** – strip surrounding whitespace from column values (default=True)
- **names** – list of names corresponding to each data column
- **include_names** – list of names to include in output (default=None selects all names)
- **exclude_names** – list of names to exlude from output (applied after include_names)

Classes

AASTex(**kwargs)	Write and read AASTeX tables.
AllType	
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 head
BaseSplitter	Base splitter that uses python's split method to do the work.
Basic()	Read a character-delimited table with a single header line at the top followed by da
Cds([readme])	Read a CDS format table.
Column(name, index)	Table column.
CommentedHeader()	Read a file where the column names are given in a line that begins with the header
ContinuationLinesInputter	Inputter where lines ending in continuation_char are joined with the subsequen
Daophot()	Read a DAOphot file.
DefaultSplitter()	Default class to split strings into columns using python csv.
FixedWidth([col_starts, col_ends, ...])	Read or write a fixed width table with a single header line that defines column nam
FixedWidthData()	Base table data reader.

Table 1.114 – continued from previous page

FixedWidthHeader()	Fixed width table header reader.
FixedWidthNoHeader([col_starts, col_ends, ...])	Read or write a 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, ...])	Read or write a fixed width table which has two header lines.
FloatType	
InconsistentTableError	
IntType	
Ipac([definition])	Read an IPAC format table.
Latex([ignore_latex_commands, latexdict, ...])	Write and read LaTeX tables.
NoHeader()	Read a table with no header line.
NoType	
NumType	
Rdb()	Read a tab-separated file with an extra line after the column definition line.
SExtractor()	Read a SExtractor file.
StrType	
Tab()	Read a tab-separated file.
TableOutputter	Output the table as an astropy.table.Table object.
WhitespaceSplitter()	

AASTex

class astropy.io.ascii.latex.AASTex(***kwargs*)

Bases: astropy.io.ascii.latex.Latex

Write and read AASTeX tables.

This class implements some AASTeX specific commands. AASTeX is used for the AAS (American Astronomical Society) publications like ApJ, ApJL and AJ.

It derives from the Latex reader and accepts the same keywords. However, the keywords header_start, header_end, data_start and data_end in latexdict have no effect.

AllType

class astropy.io.ascii.core.AllType

Bases: astropy.io.ascii.core.StrType, astropy.io.ascii.core.FloatType, astropy.io.ascii.core.IntType

BaseData

class astropy.io.ascii.core.BaseData

Bases: object

Base table data reader.

Parameters

- **start_line** – None, int, or a function of lines that returns None or int
- **end_line** – None, int, or a function of lines that returns None or int
- **comment** – Regular expression for comment lines
- **splitter_class** – Splitter class for splitting data lines into columns

Attributes Summary

comment	
fill_values	list() -> new empty list
start_line	
write_spacer_lines	list() -> new empty list
end_line	
formats	
fill_exclude_names	
fill_include_names	

Methods Summary

process_lines(lines)	Strip out comment lines and blank lines from list of lines
masks(cols)	Set fill value for each column and then apply that fill value In the first step it is evaluated with value from f
get_data_lines(lines)	Set the data_lines attribute to the lines slice comprising the table data values.
get_str_vals()	Return a generator that returns a list of column values (as strings) for each data line.
write(lines)	

Attributes Documentation

comment = **None**

fill_values = **[]**

start_line = **None**

write_spacer_lines = **['ASCII_TABLE_WRITE_SPACER_LINE']**

end_line = **None**

formats = **{}**

fill_exclude_names = **None**

fill_include_names = **None**

Methods Documentation

process_lines(*lines*)

Strip out comment lines and blank lines from list of lines

Parameters

lines – all lines in table

Returns

list of lines

masks(*cols*)
Set fill value for each column and then apply that fill value

In the first step it is evaluated with value from `fill_values` applies to which column using `fill_include_names` and `fill_exclude_names`. In the second step all replacements are done for the appropriate columns.

get_data_lines(*lines*)
Set the `data_lines` attribute to the lines slice comprising the table data values.

get_str_vals()
Return a generator that returns a list of column values (as strings) for each data line.

write(*lines*)

BaseHeader

class astropy.io.ascii.core.BaseHeader
Bases: object

Base table header reader

Parameters

- **auto_format** – format string for auto-generating column names
- **start_line** – None, int, or a function of `lines` that returns None or int
- **comment** – regular expression for comment lines
- **splitter_class** – Splitter class for splitting data lines into columns
- **names** – list of names corresponding to each data column
- **include_names** – list of names to include in output (default=None selects all names)
- **exclude_names** – list of names to exlude from output (applied after `include_names`)

Attributes Summary

comment	
exclude_names	
names	
colnames	Return the column names of the table
start_line	
include_names	
n_data_cols	Return the number of expected data columns from data splitting.
write_spacer_lines	list() -> new empty list
auto_format	str(object) -> string

Methods Summary

get_type_map_key(col)	
process_lines(lines)	Generator to yield non-comment lines

Continued on next page

Table 1.118 – continued from previous page

get_cols(lines)	Initialize the header Column objects from the table lines.
get_col_type(col)	
write(lines)	

Attributes Documentation

comment = **None**

exclude_names = **None**

names = **None**

colnames
: Return the column names of the table

start_line = **None**

include_names = **None**

n_data_cols
: Return the number of expected data columns from data splitting. This is either explicitly set (typically for fixedwidth splitters) or set to self.names otherwise.

write_spacer_lines = **['ASCII_TABLE_WRITE_SPACER_LINE']**

auto_format = **'col%d'**

Methods Documentation

get_type_map_key(*col*)

process_lines(*lines*)
: Generator to yield non-comment lines

get_cols(*lines*)
: Initialize the header Column objects from the table lines.

  Based on the previously set Header attributes find or create the column names. Sets self.cols with the list of Columns. This list only includes the actual requested columns after filtering by the include_names and exclude_names attributes. See self.names for the full list.

  Parameters
  : **lines** – list of table lines

  Returns
  : None

get_col_type(*col*)

write(*lines*)

BaseInputter

class astropy.io.ascii.core.BaseInputter

Bases: object

Get the lines from the table input and return a list of lines. The input table can be one of:

•File name

•String (newline separated) with all header and data lines (must have at least 2 lines)

•File-like object with read() method

•List of strings

Methods Summary

process_lines(lines)	Process lines for subsequent use.
get_lines(table)	Get the lines from the table input.

Methods Documentation

process_lines(*lines*)

Process lines for subsequent use. In the default case do nothing. This routine is not generally intended for removing comment lines or stripping whitespace. These are done (if needed) in the header and data line processing.

Override this method if something more has to be done to convert raw input lines to the table rows. For example the ContinuationLinesInputter derived class accounts for continuation characters if a row is split into lines.

get_lines(*table*)

Get the lines from the table input.

Parameters
table – table input

Returns
list of lines

BaseOutputter

class astropy.io.ascii.core.BaseOutputter

Bases: object

Output table as a dict of column objects keyed on column name. The table data are stored as plain python lists within the column objects.

Attributes Summary

converters

Attributes Documentation

converters = {}

BaseReader

class astropy.io.ascii.core.BaseReader

Bases: object

Class providing methods to read and write an ASCII table using the specified header, data, inputter, and outputter instances.

Typical usage is to instantiate a Reader() object and customize the header, data, inputter, and outputter attributes. Each of these is an object of the corresponding class.

There is one method inconsistent_handler that can be used to customize the behavior of read() in the event that a data row doesn't match the header. The default behavior is to raise an InconsistentTableError.

Attributes Summary

comment_lines	Return lines in the table that match header.comment regexp

Methods Summary

write(table)	Write table as list of strings.
inconsistent_handler(str_vals, ncols)	Adjust or skip data entries if a row is inconsistent with the header.
read(table)	Read the table and return the results in a format determined by the outputter attribute.

Attributes Documentation

comment_lines

Return lines in the table that match header.comment regexp

Methods Documentation

write(*table*)

Write table as list of strings.

Parameters

table – input table data (astropy.table.Table object)

Returns

list of strings corresponding to ASCII table

inconsistent_handler(*str_vals*, *ncols*)

Adjust or skip data entries if a row is inconsistent with the header.

The default implementation does no adjustment, and hence will always trigger an exception in read() any time the number of data entries does not match the header.

Note that this will *not* be called if the row already matches the header.

Parameters

- **str_vals** – A list of value strings from the current row of the table.
- **ncols** – The expected number of entries from the table header.

Returns

list of strings to be parsed into data entries in the output table. If the length of this list does not match `ncols`, an exception will be raised in read(). Can also be None, in which case the row will be skipped.

`read(`*table*`)`

Read the `table` and return the results in a format determined by the `outputter` attribute.

The `table` parameter is any string or object that can be processed by the instance `inputter`. For the base Inputter class `table` can be one of:

- File name
- String (newline separated) with all header and data lines (must have at least 2 lines)
- List of strings

Parameters

table – table input

Returns

output table

BaseSplitter

class `astropy.io.ascii.core.BaseSplitter`

Bases: `object`

Base splitter that uses python's split method to do the work.

This does not handle quoted values. A key feature is the formulation of __call__ as a generator that returns a list of the split line values at each iteration.

There are two methods that are intended to be overridden, first `process_line()` to do pre-processing on each input line before splitting and `process_val()` to do post-processing on each split string value. By default these apply the string `strip()` function. These can be set to another function via the instance attribute or be disabled entirely, for example:

```
reader.header.splitter.process_val = lambda x: x.lstrip()
reader.data.splitter.process_val = None
```

Parameters

delimiter – one-character string used to separate fields

Attributes Summary

`delimiter`

Methods Summary

process_line(line)	Remove whitespace at the beginning or end of line.
process_val(val)	Remove whitespace at the beginning or end of value.
join(vals)	

Attributes Documentation

delimiter = **None**

Methods Documentation

process_line(*line*)
: Remove whitespace at the beginning or end of line. This is especially useful for whitespace-delimited files to prevent spurious columns at the beginning or end.

process_val(*val*)
: Remove whitespace at the beginning or end of value.

join(*vals*)

Basic

class astropy.io.ascii.basic.Basic
: Bases: astropy.io.ascii.core.BaseReader

Read a character-delimited table with a single header line at the top followed by data lines to the end of the table. Lines beginning with # as the first non-whitespace character are comments. This reader is highly configurable.

```
rdr = ascii.get_reader(Reader=ascii.Basic)
rdr.header.splitter.delimiter = ' '
rdr.data.splitter.delimiter = ' '
rdr.header.start_line = 0
rdr.data.start_line = 1
rdr.data.end_line = None
rdr.header.comment = r'\s*#'
rdr.data.comment = r'\s*#'
```

Example table:

```
# Column definition is the first uncommented line
# Default delimiter is the space character.
apples oranges pears

# Data starts after the header column definition, blank lines ignored
1 2 3
4 5 6
```

Cds

class astropy.io.ascii.cds.Cds(*readme=None*)
: Bases: astropy.io.ascii.core.BaseReader

Read a CDS format table. See http://vizier.u-strasbg.fr/doc/catstd.htx. Example:

```
Table: Table name here
= ==============================================================================
Catalog reference paper
    Bibliography info here
==============================================================================
ADC_Keywords: Keyword ; Another keyword ; etc

Description:
    Catalog description here.
==============================================================================
Byte-by-byte Description of file: datafile3.txt
------------------------------------------------------------------------------
   Bytes Format Units  Label  Explanations
------------------------------------------------------------------------------
   1-  3 I3     ---    Index  Running identification number
   5-  6 I2     h      RAh    Hour of Right Ascension (J2000)
   8-  9 I2     min    RAm    Minute of Right Ascension (J2000)
  11- 15 F5.2   s      RAs    Second of Right Ascension (J2000)
------------------------------------------------------------------------------
Note (1): A CDS file can contain sections with various metadata.
          Notes can be multiple lines.
Note (2): Another note.
------------------------------------------------------------------------------
  1 03 28 39.09
  2 04 18 24.11
```

About parsing the CDS format

The CDS format consists of a table description and the table data. These can be in separate files as a ReadMe file plus data file(s), or combined in a single file. Different subsections within the description are separated by lines of dashes or equal signs ("——" or "======"). The table which specifies the column information must be preceded by a line starting with "Byte-by-byte Description of file:".

In the case where the table description is combined with the data values, the data must be in the last section and must be preceded by a section delimiter line (dashes or equal signs only).

Basic usage

Use the `ascii.read()` function as normal, with an optional `readme` parameter indicating the CDS ReadMe file. If not supplied it is assumed that the header information is at the top of the given table. Examples:

```
>>> from astropy.io import ascii
>>> table = ascii.read("t/cds.dat")
>>> table = ascii.read("t/vizier/table1.dat", readme="t/vizier/ReadMe")
>>> table = ascii.read("t/cds/multi/lhs2065.dat", readme="t/cds/multi/ReadMe")
>>> table = ascii.read("t/cds/glob/lmxbrefs.dat", readme="t/cds/glob/ReadMe")
```

Using a reader object

When `Cds` reader object is created with a `readme` parameter passed to it at initialization, then when the `read` method is executed with a table filename, the header information for the specified table is taken from the `readme` file. An `InconsistentTableError` is raised if the `readme` file does not have header information for the given table.

```
>>> readme = "t/vizier/ReadMe"
>>> r = ascii.get_reader(ascii.Cds, readme=readme)
>>> table = r.read("t/vizier/table1.dat")
>>> # table5.dat has the same ReadMe file
>>> table = r.read("t/vizier/table5.dat")
```

If no readme parameter is specified, then the header information is assumed to be at the top of the given table.

```
>>> r = ascii.get_reader(ascii.Cds)
>>> table = r.read("t/cds.dat")
>>> #The following gives InconsistentTableError, since no
>>> #readme file was given and table1.dat does not have a header.
>>> table = r.read("t/vizier/table1.dat")
Traceback (most recent call last):
  ...
InconsistentTableError: No CDS section delimiter found
```

Caveats:

- Format, Units, and Explanations are available in the Reader.cols attribute.
- All of the other metadata defined by this format is ignored.

Code contribution to enhance the parsing to include metadata in a Reader.meta attribute would be welcome.

Methods Summary

write([table])	Not available for the Cds class (raises NotImplementedError)

Methods Documentation

write(*table=None*)
Not available for the Cds class (raises NotImplementedError)

Column

class astropy.io.ascii.core.Column(*name*, *index*)
Bases: object

Table column.

The key attributes of a Column object are:

- **name** : column name
- **index** : column index (first column has index=0, second has index=1, etc)
- **type** : column type (NoType, StrType, NumType, FloatType, IntType)
- **str_vals** : list of column values as strings
- **data** : list of converted column values

CommentedHeader

class astropy.io.ascii.basic.CommentedHeader
Bases: astropy.io.ascii.core.BaseReader

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
```

ContinuationLinesInputter

class astropy.io.ascii.core.ContinuationLinesInputter

Bases: astropy.io.ascii.core.BaseInputter

Inputter where lines ending in continuation_char are joined with the subsequent line. Example:

```
col1 col2 col3
1       2 3
4 5       6
```

Attributes Summary

no_continue	
replace_char	str(object) -> string
continuation_char	str(object) -> string

Methods Summary

process_lines(lines)

Attributes Documentation

no_continue = **None**

replace_char = ' '

continuation_char = '\\'

Methods Documentation

process_lines(*lines*)

Daophot

class astropy.io.ascii.daophot.Daophot

Bases: astropy.io.ascii.core.BaseReader

Read a DAOphot file. Example:

```
#K MERGERAD   = INDEF                   scaleunit  %-23.7g
#K IRAF = NOAO/IRAFV2.10EXPORT version %-23s
#K USER = davis name %-23s
#K HOST = tucana computer %-23s
#
#N ID    XCENTER   YCENTER   MAG         MERR          MSKY           NITER    \
#U ##    pixels    pixels    magnitudes  magnitudes    counts         ##       \
#F %-9d  %-10.3f   %-10.3f   %-12.3f     %-14.3f       %-15.7g        %-6d
#
```

```
#N         SHARPNESS   CHI         PIER  PERROR                              \
#U         ##          ##          ##    perrors                             \
#F         %-23.3f     %-12.3f     %-6d  %-13s
#
14       138.538   256.405   15.461      0.003         34.85955       4        \
-0.032      0.802       0     No_error
```

The keywords defined in the #K records are available via output table `meta` attribute:

```
data = ascii.read('t/daophot.dat')
for keyword in data.meta['keywords']:
    print keyword['name'], keyword['value'], keyword['units'], keyword['format']
```

The units and formats are available in the output table columns:

```
for colname in data.colnames:
     col = data[colname]
     print colname, col.units, col.format
```

Methods Summary

`read`(table)	
`write`([table])	

Methods Documentation

`read`(*table*)

`write`(*table=None*)

DefaultSplitter

class `astropy.io.ascii.core.DefaultSplitter`

Bases: `astropy.io.ascii.core.BaseSplitter`

Default class to split strings into columns using python csv. The class attributes are taken from the csv Dialect class.

Typical usage:

```
# lines = ..
splitter = ascii.DefaultSplitter()
for col_vals in splitter(lines):
    for col_val in col_vals:
         ...
```

Parameters

- **delimiter** – one-character string used to separate fields.
- **doublequote** – control how instances of *quotechar* in a field are quoted
- **escapechar** – character to remove special meaning from following character

•**quotechar** – one-character stringto quote fields containing special characters

•**quoting** – control when quotes are recognised by the reader

•**skipinitialspace** – ignore whitespace immediately following the delimiter

Attributes Summary

escapechar	
skipinitialspace	bool(x) -> bool
quoting	int(x[, base]) -> integer
delimiter	str(object) -> string
doublequote	bool(x) -> bool
quotechar	str(object) -> string

Methods Summary

join(vals)	
process_line(line)	Remove whitespace at the beginning or end of line.

Attributes Documentation

escapechar = **None**

skipinitialspace = **True**

quoting = **0**

delimiter = **' '**

doublequote = **True**

quotechar = **'"'**

Methods Documentation

join(*vals*)

process_line(*line*)

Remove whitespace at the beginning or end of line. This is especially useful for whitespace-delimited files to prevent spurious columns at the beginning or end. If splitting on whitespace then replace unquoted tabs with space first

FixedWidth

class astropy.io.ascii.fixedwidth.FixedWidth(*col_starts=None*, *col_ends=None*, *delimiter_pad=' '*, *bookend=True*)

Bases: astropy.io.ascii.core.BaseReader

Read or write a fixed width table with a single header line that defines column names and positions. Examples:

```
# Bar delimiter in header and data

|  Col1 |   Col2      |  Col3 |
|  1.2  | hello there |     3 |
|  2.4  | many words  |     7 |

# Bar delimiter in header only

Col1 |   Col2      | Col3
1.2    hello there    3
2.4    many words     7

# No delimiter with column positions specified as input

Col1       Col2Col3
 1.2hello there   3
 2.4many words    7
```

See the *Fixed-width Gallery* for specific usage examples.

Parameters

- **col_starts** – list of start positions for each column (0-based counting)
- **col_ends** – list of end positions (inclusive) for each column
- **delimiter_pad** – padding around delimiter when writing (default = None)
- **bookend** – put the delimiter at start and end of line when writing (default = False)

FixedWidthData

class astropy.io.ascii.fixedwidth.FixedWidthData

Bases: astropy.io.ascii.core.BaseData

Base table data reader.

Parameters

- **start_line** – None, int, or a function of `lines` that returns None or int
- **end_line** – None, int, or a function of `lines` that returns None or int
- **comment** – Regular expression for comment lines
- **splitter_class** – Splitter class for splitting data lines into columns

Methods Summary

write(lines)	

Methods Documentation

write(*lines*)

FixedWidthHeader
class astropy.io.ascii.fixedwidth.FixedWidthHeader
Bases: astropy.io.ascii.core.BaseHeader

Fixed width table header reader.

The key settable class attributes are:

Parameters

- **auto_format** – format string for auto-generating column names
- **start_line** – None, int, or a function of lines that returns None or int
- **comment** – regular expression for comment lines
- **splitter_class** – Splitter class for splitting data lines into columns
- **names** – list of names corresponding to each data column
- **include_names** – list of names to include in output (default=None selects all names)
- **exclude_names** – list of names to exlude from output (applied after include_names)
- **position_line** – row index of line that specifies position (default = 1)
- **position_char** – character used to write the position line (default = "-")
- **col_starts** – list of start positions for each column (0-based counting)
- **col_ends** – list of end positions (inclusive) for each column
- **delimiter_pad** – padding around delimiter when writing (default = None)
- **bookend** – put the delimiter at start and end of line when writing (default = False)

Attributes Summary

position_line

Methods Summary

write(lines)	
get_cols(lines)	Initialize the header Column objects from the table lines.
get_line(lines, index)	
get_fixedwidth_params(line)	Split line on the delimiter and determine column values and column start and end positions.

Attributes Documentation

position_line = **None**

Methods Documentation

write(*lines*)

get_cols(*lines*)

Initialize the header Column objects from the table `lines`.

Based on the previously set Header attributes find or create the column names. Sets `self.cols` with the list of Columns. This list only includes the actual requested columns after filtering by the include_names and exclude_names attributes. See `self.names` for the full list.

Parameters
lines – list of table lines

Returns
None

get_line(*lines*, *index*)

get_fixedwidth_params(*line*)

Split `line` on the delimiter and determine column values and column start and end positions. This might include null columns with zero length (e.g. for `header row = "| col1 || col2 | col3 |"` or `header2_row = "----- ------- -----"`). The null columns are stripped out. Returns the values between delimiters and the corresponding start and end positions.

Parameters
line – input line

Returns
(vals, starts, ends)

FixedWidthNoHeader

class astropy.io.ascii.fixedwidth.FixedWidthNoHeader(*col_starts=None*, *col_ends=None*, *delimiter_pad=' '*, *bookend=True*)

Bases: astropy.io.ascii.fixedwidth.FixedWidth

Read or write a fixed width table which has no header line. Column names are either input (`names` keyword) or auto-generated. Column positions are determined either by input (`col_starts` and `col_stops` keywords) or by splitting the first data line. In the latter case a `delimiter` is required to split the data line.

Examples:

```
# Bar delimiter in header and data

|  1.2  | hello there |     3 |
|  2.4  | many words  |     7 |

# Compact table having no delimiter and column positions specified as input

1.2hello there3
2.4many words 7
```

This class is just a convenience wrapper around the `FixedWidth` reader but with `header.start_line = None` and `data.start_line = 0`.

See the *Fixed-width Gallery* for specific usage examples.

Parameters

- **col_starts** – list of start positions for each column (0-based counting)
- **col_ends** – list of end positions (inclusive) for each column
- **delimiter_pad** – padding around delimiter when writing (default = None)
- **bookend** – put the delimiter at start and end of line when writing (default = False)

FixedWidthSplitter

class astropy.io.ascii.fixedwidth.FixedWidthSplitter

Bases: astropy.io.ascii.core.BaseSplitter

Split line based on fixed start and end positions for each col in self.cols.

This class requires that the Header class will have defined col.start and col.end for each column. The reference to the header.cols gets put in the splitter object by the base Reader.read() function just in time for splitting data lines by a data object.

Note that the start and end positions are defined in the pythonic style so line[start:end] is the desired substring for a column. This splitter class does not have a hook for process_lines since that is generally not useful for fixed-width input.

Attributes Summary

delimiter_pad	str(object) -> string
bookend	bool(x) -> bool

Methods Summary

join(vals, widths)

Attributes Documentation

delimiter_pad = ''

bookend = False

Methods Documentation

join(*vals*, *widths*)

FixedWidthTwoLine

class astropy.io.ascii.fixedwidth.FixedWidthTwoLine(*position_line=1*, *position_char='-'*, *delimiter_pad=None*, *bookend=False*)

Bases: astropy.io.ascii.fixedwidth.FixedWidth

Read or write a fixed width table which has two header lines. The first header line defines the column names and the second implicitly defines the column positions. Examples:

```
# Typical case with column extent defined by ---- under column names.

 col1    col2         <== header_start = 0
-----  ------------   <== position_line = 1, position_char = "-"
  1     bee flies     <== data_start = 2
  2     fish swims

# Pretty-printed table

+------+------------+
| Col1 |   Col2     |
+------+------------+
|  1.2 | "hello"    |
|  2.4 | there world|
+------+------------+
```

See the *Fixed-width Gallery* for specific usage examples.

Parameters

- **position_line** – row index of line that specifies position (default = 1)
- **position_char** – character used to write the position line (default = "-")
- **delimiter_pad** – padding around delimiter when writing (default = None)
- **bookend** – put the delimiter at start and end of line when writing (default = False)

FloatType
class astropy.io.ascii.core.FloatType
Bases: astropy.io.ascii.core.NumType

InconsistentTableError
exception astropy.io.ascii.core.InconsistentTableError

IntType
class astropy.io.ascii.core.IntType
Bases: astropy.io.ascii.core.NumType

Ipac
class astropy.io.ascii.ipac.Ipac(*definition='ignore'*)
Bases: astropy.io.ascii.core.BaseReader

Read an IPAC format table. See http://irsa.ipac.caltech.edu/applications/DDGEN/Doc/ipac_tbl.html:

```
\name=value
\ Comment
|  column1 |  column2 | column3 | column4  |    column5       |
|  double  |  double  |   int   |   double |     char         |
|   unit   |   unit   |   unit  |    unit  |     unit         |
|   null   |   null   |   null  |    null  |     null         |
 2.0978     29.09056   73765     2.06000    B8IVpMnHg
```

Or:

```
|-----ra---|----dec---|---sao---|------v---|----sptype--------|
  2.09708   29.09056     73765    2.06000   B8IVpMnHg
```

Parameters

definition : str, optional

Specify the convention for characters in the data table that occur directly below the pipe (|) symbol in the header column definition:

- 'ignore' - Any character beneath a pipe symbol is ignored (default)
- 'right' - Character is associated with the column to the right
- 'left' - Character is associated with the column to the left

Notes

Caveats:

- Data type, Units, and Null value specifications are ignored.
- Keywords are ignored.
- The IPAC spec requires the first two header lines but this reader only requires the initial column name definition line

Overcoming these limitations would not be difficult, code contributions welcome from motivated users.

Methods Summary

write([table])	Not available for the Ipac class (raises NotImplementedError)

Methods Documentation

write(*table=None*)

Not available for the Ipac class (raises NotImplementedError)

Latex

class astropy.io.ascii.latex.Latex(*ignore_latex_commands=['hline', 'vspace', 'tableline'], latexdict={}, caption='', col_align=None*)

Bases: astropy.io.ascii.core.BaseReader

Write and read LaTeX tables.

This class implements some LaTeX specific commands. Its main purpose is to write out a table in a form that LaTeX can compile. It is beyond the scope of this class to implement every possible LaTeX command, instead the focus is to generate a syntactically valid LaTeX tables.

This class can also read simple LaTeX tables (one line per table row, no \multicolumn or similar constructs), specifically, it can read the tables that it writes.

Reading a LaTeX table, the following keywords are accepted:

ignore_latex_commands :

Lines starting with these LaTeX commands will be treated as comments (i.e. ignored).

When writing a LaTeX table, the some keywords can customize the format. Care has to be taken here, because python interprets \\ in a string as an escape character. In order to pass this to the output either format your strings as raw strings with the `r` specifier or use a double \\\\.

Examples:

```
caption = r'My table \label{mytable}'
caption = 'My table \\\\label{mytable}'
```

latexdict : Dictionary of extra parameters for the LaTeX output

- **tabletype**
  [used for first and last line of table.] The default is `\\begin{table}`. The following would generate a table, which spans the whole page in a two-column document:

  ```
  ascii.write(data, sys.stdout, Writer = ascii.Latex,
              latexdict = {'tabletype': 'table*'})
  ```

- **col_align**
  [Alignment of columns] If not present all columns will be centered.

- **caption**
  [Table caption (string or list of strings)] This will appear above the table as it is the standard in many scientific publications. If you prefer a caption below the table, just write the full LaTeX command as `latexdict['tablefoot'] = r'\caption{My table}'`

- **preamble, header_start, header_end, data_start, data_end, tablefoot: Pure LaTeX**
  Each one can be a string or a list of strings. These strings will be inserted into the table without any further processing. See the examples below.

- **units**
  [dictionary of strings] Keys in this dictionary should be names of columns. If present, a line in the LaTeX table directly below the column names is added, which contains the values of the dictionary. Example:

  ```
  from astropy.io import ascii
  data = {'name': ['bike', 'car'], 'mass': [75,1200], 'speed': [10, 130]}
  ascii.write(data, Writer=ascii.Latex,
                  latexdict = {'units': {'mass': 'kg', 'speed': 'km/h'}})
  ```

  If the column has no entry in the `units` dictionary, it defaults to `' '`.

Run the following code to see where each element of the dictionary is inserted in the LaTeX table:

```
from astropy.io import ascii
data = {'cola': [1,2], 'colb': [3,4]}
ascii.write(data, Writer=ascii.Latex, latexdict=ascii.latex.latexdicts['template'])
```

Some table styles are predefined in the dictionary `ascii.latex.latexdicts`. The following generates in table in style preferred by A&A and some other journals:

```
ascii.write(data, Writer=ascii.Latex, latexdict=ascii.latex.latexdicts['AA'])
```

As an example, this generates a table, which spans all columns and is centered on the page:

```
ascii.write(data, Writer=ascii.Latex, col_align='|lr|',
            latexdict={'preamble': r'\begin{center}',
                       'tablefoot': r'\end{center}',
                       'tabletype': 'table*'})
```

caption
[Set table caption] Shorthand for:

```
latexdict['caption'] = caption
```

col_align
[Set the column alignment.] If not present this will be auto-generated for centered columns. Shorthand for:

```
latexdict['col_align'] = col_align
```

Methods Summary

write([table])

Methods Documentation

write(*table=None*)

NoHeader

class astropy.io.ascii.basic.NoHeader
Bases: astropy.io.ascii.basic.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 Basic class from which it is derived. Example:

```
# Table data
1 2 "hello there"
3 4 world
```

NoType

class astropy.io.ascii.core.NoType
Bases: object

NumType

class astropy.io.ascii.core.NumType
Bases: astropy.io.ascii.core.NoType

Rdb

class astropy.io.ascii.basic.Rdb
Bases: astropy.io.ascii.basic.Tab

Read a tab-separated file with an extra line after the column definition line. The RDB format meets this definition. Example:

```
col1 <tab> col2 <tab> col3
N <tab> S <tab> N
1 <tab> 2 <tab> 5
```

In this reader the second line is just ignored.

SExtractor
class astropy.io.ascii.sextractor.SExtractor
Bases: astropy.io.ascii.core.BaseReader

Read a SExtractor file.
SExtractor is a package for faint-galaxy photometry. Bertin & Arnouts 1996, A&A Supp. 317, 393. http://www.astromatic.net/software/sextractor

Example:

```
# 1 NUMBER
# 2 ALPHA_J2000
# 3 DELTA_J2000
# 4 FLUX_RADIUS
# 7 MAG_AUTO
1 32.23222 10.1211 0.8 1.2 1.4 18.1
2 38.12321 -88.1321 2.2 2.4 3.1 17.0
```

Note the skipped numbers since flux_radius has 3 columns. The three FLUX_RADIUS columns will be named FLUX_RADIUS, FLUX_RADIUS_1, FLUX_RADIUS_2

Methods Summary

read(table)
write([table])

Methods Documentation

read(*table*)

write(*table=None*)

StrType
class astropy.io.ascii.core.StrType
Bases: astropy.io.ascii.core.NoType

Tab
class astropy.io.ascii.basic.Tab
Bases: astropy.io.ascii.basic.Basic

Read a tab-separated file. Unlike the Basic reader, whitespace is not stripped from the beginning and end of lines. By default whitespace is still stripped from the beginning and end of individual column values.

Example:

```
col1 <tab> col2 <tab> col3
# Comment line
1 <tab> 2 <tab> 5
```

TableOutputter

class astropy.io.ascii.core.TableOutputter
Bases: astropy.io.ascii.core.BaseOutputter

Output the table as an astropy.table.Table object.

Missing or bad data values are not presently handled and raise an exception. This will be changed, but in the meantime use the NumpyOutputter.

Attributes Summary

default_converters	list() -> new empty list

Attributes Documentation

default_converters = **[(<function converter at 0x41c2f50>, <class 'astropy.io.ascii.core.IntType'>), (<function converter**

WhitespaceSplitter
class astropy.io.ascii.core.WhitespaceSplitter
Bases: astropy.io.ascii.core.DefaultSplitter

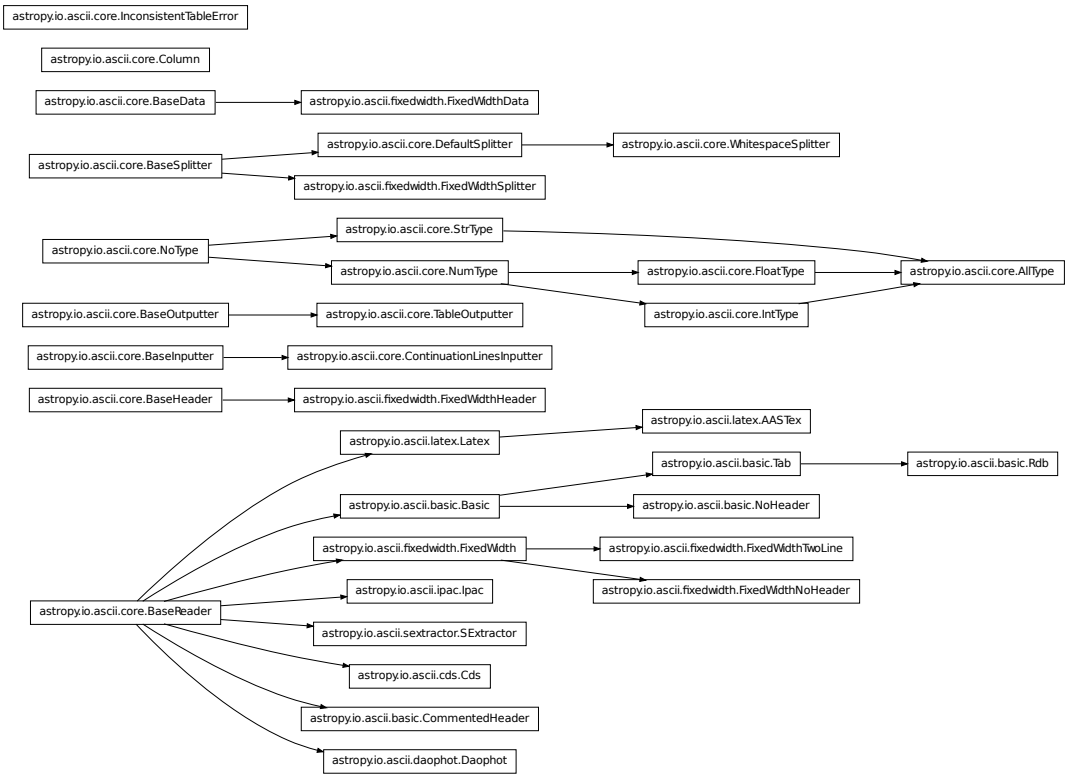
Methods Summary

process_line(line)	Replace tab with space within line while respecting quoted substrings

Methods Documentation

process_line(*line*)
Replace tab with space within line while respecting quoted substrings

Class Inheritance Diagram



1.12 VOTable XML handling (`astropy.io.votable`)

1.12.1 Introduction

The `astropy.io.votable` subpackage converts VOTable XML files to and from Numpy record arrays.

1.12.2 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, one 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, one 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 there is expected to be only one table in the file, it might be simplest to just use get_first_table:

```
table = votable.get_first_table()
```

Even easier, 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, one 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. For example, 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>
```

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:

```
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, simply 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 currently two 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. 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')
```

1.12.3 Using `io.votable`

Standard compliance

`astropy.io.votable.table` supports the VOTable Format Definition Version 1.1 and Version 1.2. Some flexibility is provided to support the 1.0 draft version and other non-standard 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 or 1.2 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 simply 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 - In Python 2.x, a `str` object; in 3.x, a `bytes` object.
char (*fixed length*)	S
unicodeChar (*variable length*)	O - In Python 2.x, a `unicode` object, in utf-16; in 3.x a `str` object
unicodeChar (*fixed length*)	U
short	i2
int	i4
long	i8
float	f4
double	f8
floatComplex	c8
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, one can use the `get_field_by_id` method, which returns the `Field` object with the given ID. For example:

```
>>> field = table.get_field_or_param_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.

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")
```

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.

1.12.4 See Also

- VOTable Format Definition Version 1.1
- VOTable Format Definition Version 1.2

1.12.5 Reference/API

astropy.io.votable Module

This package reads and writes data formats used by the Virtual Observatory (VO) initiative, particularly the VOTable XML format.

Functions

parse(source[, columns, invalid, pedantic, ...])	Parses a VOTABLE xml file (or file-like object), and returns a VOTable object.
parse_single_table(source, **kwargs)	Parses a VOTABLE xml file (or file-like object), reading and returning only the first
validate(source[, output, xmllint, filename])	Prints a validation report for the given file.
from_table(table[, table_id])	Given an astropy.table.Table object, return a VOTableFile file structure contain
is_votable(source)	Reads the header of a file to determine if it is a VOTable file.

parse

astropy.io.votable.table.parse(*source*, *columns=None*, *invalid='exception'*, *pedantic=None*, *chunk_size=256*, *table_number=None*, *table_id=None*, *filename=None*, *_debug_python_based_parser=False*)

Parses a VOTABLE xml file (or file-like object), and returns a VOTable object.

Parameters

source : str or readable file-like object

Path or file object containing a VOTABLE xml file.

columns : sequence of str, optional

List of field names to include in the output. The default is to include all fields.

invalid : str, optional

One of the following values:

- 'exception': throw an exception when an invalid value is encountered (default)
- 'mask': mask out invalid values

pedantic : bool, optional

When True, raise an error when the file violates the spec, otherwise issue a warning. Warnings may be controlled using the standard Python mechanisms. See the warnings module in the Python standard library for more information. When not

provided, uses the configuration setting `astropy.io.votable.pedantic`, which defaults to True.

chunk_size : int, optional

The number of rows to read before converting to an array. Higher numbers are likely to be faster, but will consume more memory.

table_number : int, optional

The number of table in the file to read in. If `None`, all tables will be read. If a number, 0 refers to the first table in the file, and only that numbered table will be parsed and read in. Should not be used with `table_id`.

table_id : str, optional

The ID of the table in the file to read in. Should not be used with `table_number`.

filename : str, optional

A filename, URL or other identifier to use in error messages. If *filename* is None and *source* is a string (i.e. a path), then *source* will be used as a filename for error messages. Therefore, *filename* is only required when source is a file-like object.

Returns

votable : `astropy.io.votable.tree.VOTableFile` object

See Also:

`astropy.io.votable.exceptions`
The exceptions this function may raise.

parse_single_table

`astropy.io.votable.table.parse_single_table(`*source*, ***kwargs*`)`

Parses a VOTABLE xml file (or file-like object), reading and returning only the first `Table` instance.

See `parse` for a description of the keyword arguments.

Returns

votable : `astropy.io.votable.tree.Table` object

validate

`astropy.io.votable.table.validate(`*source*, *output=<open file '<stdout>', mode 'w' at 0x7f23551061e0>*, *xmllint=False*, *filename=None*`)`

Prints a validation report for the given file.

Parameters

source : str or readable file-like object

Path to a VOTABLE xml file.

output : writable file-like object, optional

Where to output the report. Defaults to `sys.stdout`. If `None`, the output will be returned as a string.

xmllint : bool, optional

When `True`, also send the file to `xmllint` for schema and DTD validation. Requires that `xmllint` is installed. The default is `False`. `source` must be a file on the local filesystem in order for `xmllint` to work.

filename : str, optional

A filename to use in the error messages. If not provided, one will be automatically determined from `source`.

Returns

is_valid : bool or str

Returns `True` if no warnings were found. If `output` is `None`, the return value will be a string.

from_table

`astropy.io.votable.table.from_table(`*table*, *table_id=None*`)`

Given an `astropy.table.Table` object, return a `VOTableFile` file structure containing just that single table.

Parameters

table : `astropy.table.Table` instance

table_id : str, optional

If not `None`, set the given id on the returned `Table` instance.

Returns

votable : `astropy.io.votable.tree.VOTableFile` instance

is_votable

`astropy.io.votable.table.is_votable(`*source*`)`

Reads the header of a file to determine if it is a VOTable file.

Parameters

source : str or readable file-like object

Path or file object containing a VOTABLE xml file.

Returns

is_votable : bool

Returns `True` if the given file is a VOTable file.

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 o
`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.
`FieldRef`(table, ref[, ucd, utype, config, pos])	FIELDref element: used inside of GROUP elements to refer to remote FIELD eleme
`ParamRef`(table, ref[, ucd, utype, config, pos])	PARAMref element: used inside of GROUP elements to refer to remote PARAM ele
`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.

Link

class astropy.io.votable.tree.Link(*ID=None*, *title=None*, *value=None*, *href=None*, *action=None*, *id=None*, *config={}*, *pos=None*, ***kwargs*)

Bases: astropy.io.votable.tree.SimpleElement, astropy.io.votable.tree._IDProperty

LINK elements: used to reference external documents and servers through a URI.

The keyword arguments correspond to setting members of the same name, documented below.

Attributes Summary

href	A URI to an arbitrary protocol.
content_role	Defines the MIME role of the referenced object.
content_type	Defines the MIME content type of the referenced object.

Methods Summary

to_table_column(column)
from_table_column(d)

Attributes Documentation

href
: A URI to an arbitrary protocol. The vo package only supports http and anonymous ftp.

content_role
: Defines the MIME role of the referenced object. Must be one of:

  None, 'query', 'hints', 'doc' or 'location'

content_type
: Defines the MIME content type of the referenced object.

Methods Documentation

to_table_column(*column*)

classmethod from_table_column(*d*)

Info

class astropy.io.votable.tree.Info(*ID=None*, *name=None*, *value=None*, *id=None*, *xtype=None*, *ref=None*, *unit=None*, *ucd=None*, *utype=None*, *config={}*, *pos=None*, ***extra*)

Bases: astropy.io.votable.tree.SimpleElementWithContent, astropy.io.votable.tree._IDProperty, astropy.io.votable.tree._XtypeProperty, astropy.io.votable.tree._UtypeProperty

INFO elements: arbitrary key-value pairs for extensions to the standard.

The keyword arguments correspond to setting members of the same name, documented below.

Attributes Summary

name	[*required*] The key of the key-value pair.
value	[*required*] The value of the key-value pair. (Always stored
content	The content inside the INFO element.
ref	Refer to another INFO element by ID, defined previously in the document.
unit	A string specifying the units for the INFO.

Methods Summary

to_xml(w, **kwargs)

Attributes Documentation

name
: [*required*] The key of the key-value pair.

value
: [*required*] The value of the key-value pair. (Always stored as a string or unicode string).

content
: The content inside the INFO element.

ref
: Refer to another INFO element by ID, defined previously in the document.

unit
: A string specifying the units for the INFO.

Methods Documentation

to_xml(*w*, ***kwargs*)

Values

class astropy.io.votable.tree.Values(*votable*, *field*, *ID=None*, *null=None*, *ref=None*, *type='legal'*, *id=None*, *config={}*, *pos=None*, ***extras*)

Bases: astropy.io.votable.tree.Element, astropy.io.votable.tree._IDProperty

VALUES element: used within FIELD and PARAM elements to define the domain of values.

The keyword arguments correspond to setting members of the same name, documented below.

Attributes Summary

max	The maximum value of the domain.
min_inclusive	When True, the domain includes the minimum value.
null	For integral datatypes, *null* is used to define the value used for missing values.
max_inclusive	When True, the domain includes the maximum value.

Continued on next pag

Table 1.147 – continued from previous page

min	The minimum value of the domain.
ref	Refer to another VALUES element by ID, defined previously in the document, for MIN/MAX/OPTION informatio
type	[*required*] Defines the applicability of the domain defined
options	A list of string key-value tuples defining other OPTION elements for the domain.

Methods Summary

parse(iterator, config)
to_xml(w, **kwargs)
to_table_column(column)
from_table_column(column)
is_defaults()

Attributes Documentation

`max`
The maximum value of the domain. See `max_inclusive`.

`min_inclusive`
When `True`, the domain includes the minimum value.

`null`
For integral datatypes, *null* is used to define the value used for missing values.

`max_inclusive`
When `True`, the domain includes the maximum value.

`min`
The minimum value of the domain. See `min_inclusive`.

`ref`
Refer to another VALUES element by ID, defined previously in the document, for MIN/MAX/OPTION information.

`type`
[*required*] Defines the applicability of the domain defined by this VALUES element. Must be one of the following strings:

- 'legal': The domain of this column applies in general to this datatype. (default)
- 'actual': The domain of this column applies only to the data enclosed in the parent table.

`options`
A list of string key-value tuples defining other OPTION elements for the domain. All options are ignored – they are stored for round-tripping purposes only.

Methods Documentation

`parse(`*iterator*, *config*`)`

`to_xml(`*w*, ***kwargs*`)`

to_table_column(*column*)

from_table_column(*column*)

is_defaults()

Field

class astropy.io.votable.tree.Field(*votable*, *ID=None*, *name=None*, *datatype=None*, *arraysize=None*, *ucd=None*, *unit=None*, *width=None*, *precision=None*, *utype=None*, *ref=None*, *type=None*, *id=None*, *xtype=None*, *config={}*, *pos=None*, ***extra*)

Bases: astropy.io.votable.tree.SimpleElement, astropy.io.votable.tree._IDProperty, astropy.io.votable.tree._NameProperty, astropy.io.votable.tree._XtypeProperty, astropy.io.votable.tree._UtypeProperty, astropy.io.votable.tree._UcdProperty

FIELD element: describes the datatype of a particular column of data.

The keyword arguments correspond to setting members of the same name, documented below.

If *ID* is provided, it is used for the column name in the resulting recarray of the table. If no *ID* is provided, *name* is used instead. If neither is provided, an exception will be raised.

Attributes Summary

links	A list of Link instances used to reference more details about the meaning of the FIELD.
precision	Along with width, defines the numerical accuracy associated with the data.
arraysize	Specifies the size of the multidimensional array if this FIELD contains more than a single value.
unit	A string specifying the units for the FIELD.
datatype	[*required*] The datatype of the column. Valid values (as
type	The type attribute on FIELD elements is reserved for future extensions.
width	Along with precision, defines the numerical accuracy associated with the data.
values	A Values instance (or None) defining the domain of the column.
ref	On FIELD elements, ref is used only for informational purposes, for example to refer to a COOSYS element.

Methods Summary

parse(iterator, config)	
to_xml(w, **kwargs)	
to_table_column(column)	Sets the attributes of a given astropy.table.Column instance to match the information in t
from_table_column(votable, column)	Restores a Field instance from a given astropy.table.Column instance.
uniqify_names(fields)	Make sure that all names and titles in a list of fields are unique, by appending numbers if nec

Attributes Documentation

links

A list of Link instances used to reference more details about the meaning of the FIELD. This is purely informational and is not used by the astropy.io.votable package.

precision

Along with width, defines the numerical accuracy associated with the data. These values are used to limit the precision when writing floating point values back to the XML file. Otherwise, it is purely informational – the Numpy recarray containing the data itself does not use this information.

arraysize
Specifies the size of the multidimensional array if this FIELD contains more than a single value.

See multidimensional arrays.

unit
A string specifying the units for the FIELD.

datatype
[*required*] The datatype of the column. Valid values (as defined by the spec) are:

'boolean', 'bit', 'unsignedByte', 'short', 'int', 'long', 'char', 'unicodeChar', 'float', 'double', 'floatComplex', or 'doubleComplex'

Many VOTABLE files in the wild use 'string' instead of 'char', so that is also a valid option, though 'string' will always be converted to 'char' when writing the file back out.

type
The type attribute on FIELD elements is reserved for future extensions.

width
Along with precision, defines the numerical accuracy associated with the data. These values are used to limit the precision when writing floating point values back to the XML file. Otherwise, it is purely informational – the Numpy recarray containing the data itself does not use this information.

values
A Values instance (or None) defining the domain of the column.

ref
On FIELD elements, ref is used only for informational purposes, for example to refer to a COOSYS element.

Methods Documentation

parse(*iterator*, *config*)

to_xml(*w*, ***kwargs*)

to_table_column(*column*)
Sets the attributes of a given astropy.table.Column instance to match the information in this Field.

classmethod from_table_column(*votable*, *column*)
Restores a Field instance from a given astropy.table.Column instance.

classmethod uniqify_names(*fields*)
Make sure that all names and titles in a list of fields are unique, by appending numbers if necessary.

Param

class astropy.io.votable.tree.Param(*votable*, *ID=None*, *name=None*, *value=None*, *datatype=None*, *arraysize=None*, *ucd=None*, *unit=None*, *width=None*, *precision=None*, *utype=None*, *type=None*, *id=None*, *config={}*, *pos=None*, ***extra*)

Bases: astropy.io.votable.tree.Field

PARAM element: constant-valued columns in the data.

`Param` objects are a subclass of `Field`, and have all of its methods and members. Additionally, it defines `value`.

Attributes Summary

`value`	[*required*] The constant value of the parameter. Its type is

Methods Summary

`to_xml`(w, **kwargs)

Attributes Documentation

`value`
: [*required*] The constant value of the parameter. Its type is determined by the `datatype` member.

Methods Documentation

`to_xml`(*w*, ***kwargs*)

CooSys

class `astropy.io.votable.tree.CooSys`(*ID=None*, *equinox=None*, *epoch=None*, *system=None*, *id=None*, *config={}*, *pos=None*, ***extra*)

Bases: `astropy.io.votable.tree.SimpleElement`

COOSYS element: defines a coordinate system.

The keyword arguments correspond to setting members of the same name, documented below.

Attributes Summary

`equinox`	A parameter required to fix the equatorial or ecliptic systems (as e.g.
`system`	Specifies the type of coordinate system.
`ID`	[*required*] The XML ID of the COOSYS element, used for
`epoch`	Specifies the epoch of the positions.

Attributes Documentation

`equinox`
: A parameter required to fix the equatorial or ecliptic systems (as e.g. "J2000" as the default "eq_FK5" or "B1950" as the default "eq_FK4").

`system`
: Specifies the type of coordinate system. Valid choices are:

'eq_FK4', 'eq_FK5', 'ICRS', 'ecl_FK4', 'ecl_FK5', 'galactic', 'supergalactic', 'xy',

'barycentric', or 'geo_app'

`ID`
: [*required*] The XML ID of the COOSYS element, used for cross-referencing. May be `None` or a string conforming to XML ID syntax.

`epoch`
: Specifies the epoch of the positions. It must be a string specifying an astronomical year.

FieldRef

class `astropy.io.votable.tree.FieldRef`(*table*, *ref*, *ucd=None*, *utype=None*, *config={}*, *pos=None*, ***extra*)

Bases: `astropy.io.votable.tree.SimpleElement`, `astropy.io.votable.tree._UtypeProperty`, `astropy.io.votable.tree._UcdProperty`

FIELDref element: used inside of GROUP elements to refer to remote FIELD elements.

Attributes Summary

`ref`	The ID of the FIELD that this FIELDref references.

Methods Summary

`get_ref`()	Lookup the `Field` instance that this `FieldRef` references.

Attributes Documentation

`ref`
: The ID of the FIELD that this FIELDref references.

Methods Documentation

`get_ref`()
: Lookup the `Field` instance that this `FieldRef` references.

ParamRef

class `astropy.io.votable.tree.ParamRef`(*table*, *ref*, *ucd=None*, *utype=None*, *config={}*, *pos=None*)

Bases: `astropy.io.votable.tree.SimpleElement`, `astropy.io.votable.tree._UtypeProperty`, `astropy.io.votable.tree._UcdProperty`

PARAMref element: used inside of GROUP elements to refer to remote PARAM elements.

The keyword arguments correspond to setting members of the same name, documented below.

It contains the following publicly-accessible members:

ref: An XML ID refering to a <PARAM> element.

Attributes Summary

ref	The ID of the PARAM that this PARAMref references.

Methods Summary

get_ref()	Lookup the Param instance that this PARAMref references.

Attributes Documentation

ref
: The ID of the PARAM that this PARAMref references.

Methods Documentation

get_ref()
: Lookup the Param instance that this PARAMref references.

Group

class astropy.io.votable.tree.Group(*table*, *ID=None*, *name=None*, *ref=None*, *ucd=None*, *utype=None*, *id=None*, *config={}*, *pos=None*, ***extra*)

Bases: astropy.io.votable.tree.Element, astropy.io.votable.tree._IDProperty, astropy.io.votable.tree._NameProperty, astropy.io.votable.tree._UtypeProperty, astropy.io.votable.tree._UcdProperty, astropy.io.votable.tree._DescriptionProperty

GROUP element: groups FIELD and PARAM elements.

This information is currently ignored by the vo package—that is the columns in the recarray are always flat—but the grouping information is stored so that it can be written out again to the XML file.

The keyword arguments correspond to setting members of the same name, documented below.

Attributes Summary

entries	[read-only] A list of members of the GROUP. This list may
ref	Currently ignored, as it's not clear from the spec how this is meant to work.

Methods Summary

parse(iterator, config)	
iter_groups()	Recursively iterate over all sub-Group instances in this Group.
to_xml(w, **kwargs)	
iter_fields_and_params()	Recursively iterate over all Param elements in this Group.

Attributes Documentation

entries
: [read-only] A list of members of the GROUP. This list may only contain objects of type Param, Group, ParamRef and FieldRef.

`ref`
Currently ignored, as it's not clear from the spec how this is meant to work.

Methods Documentation

`parse(`*iterator*, *config*`)`

`iter_groups()`
Recursively iterate over all sub-`Group` instances in this `Group`.

`to_xml(`*w*, ***kwargs*`)`

`iter_fields_and_params()`
Recursively iterate over all `Param` elements in this `Group`.

Table

class `astropy.io.votable.tree.Table(`*votable*, *ID=None*, *name=None*, *ref=None*, *ucd=None*, *utype=None*, *nrows=None*, *id=None*, *config={}*, *pos=None*, ***extra*`)`

Bases: `astropy.io.votable.tree.Element`, `astropy.io.votable.tree._IDProperty`, `astropy.io.votable.tree._NameProperty`, `astropy.io.votable.tree._UcdProperty`, `astropy.io.votable.tree._DescriptionProperty`

TABLE element: optionally contains data.

It contains the following publicly-accessible and mutable attribute:

array: A Numpy masked array of the data itself, where each row is a row of votable data, and columns are named and typed based on the <FIELD> elements of the table. The mask is parallel to the data array, except for variable-length fields. For those fields, the numpy array's column type is "object" (`"O"`), and another masked array is stored there.

If the Table contains no data, (for example, its enclosing `Resource` has `type` == 'meta') *array* will have zero-length.

The keyword arguments correspond to setting members of the same name, documented below.

Attributes Summary

`links`	A list of `Link` objects (pointers to other documents or servers through a URI) for the table.
`params`	A list of parameters (constant-valued columns) for the table.
`ref`	
`format`	[*required*] The serialization format of the table. Must be
`nrows`	[*immutable*] The number of rows in the table, as specified in
`groups`	A list of `Group` objects describing how the columns and parameters are grouped.
`infos`	A list of `Info` objects for the table.
`fields`	A list of `Field` objects describing the types of each of the data columns.

Methods Summary

Table 1.161 – continued from previous page

parse(iterator, config)	
to_table()	Convert this VO Table to an astropy.table.Table instance.
get_field_by_id(ref[, before])	Looks up a FIELD or PARAM element by the given ID.
iter_groups()	Recursively iterate over all GROUP elements in the TABLE.
create_arrays([nrows, config])	Create a new array to hold the data based on the current set of fields, and store them in th
get_group_by_id(ref[, before])	Looks up a GROUP element by the given ID.
is_empty()	Returns True if this table doesn't contain any real data because it was skipped over by th
to_xml(w, **kwargs)	
iter_fields_and_params()	Recursively iterate over all FIELD and PARAM elements in the TABLE.
get_field_by_id_or_name(ref[, before])	Looks up a FIELD or PARAM element by the given ID or name.
from_table(votable, table)	Create a Table instance from a given astropy.table.Table instance.

Attributes Documentation

links
: A list of Link objects (pointers to other documents or servers through a URI) for the table.

params
: A list of parameters (constant-valued columns) for the table. Must contain only Param objects.

ref

format
: [*required*] The serialization format of the table. Must be one of:

  'tabledata' (TABLEDATA), 'binary' (BINARY), 'fits' (FITS).

  Note that the 'fits' format, since it requires an external file, can not be written out. Any file read in with 'fits' format will be read out, by default, in 'tabledata' format.

nrows
: [*immutable*] The number of rows in the table, as specified in the XML file.

groups
: A list of Group objects describing how the columns and parameters are grouped. Currently this information is only kept around for round-tripping and informational purposes.

infos
: A list of Info objects for the table. Allows for post-operational diagnostics.

fields
: A list of Field objects describing the types of each of the data columns.

Methods Documentation

parse(*iterator*, *config*)

to_table()
: Convert this VO Table to an astropy.table.Table instance.

  Warning: Variable-length array fields may not be restored identically when round-tripping through the astropy.table.Table instance.

`get_field_by_id`(*ref*, *before=None*)
Looks up a FIELD or PARAM element by the given ID.

`iter_groups`()
Recursively iterate over all GROUP elements in the TABLE.

`create_arrays`(*nrows=0*, *config={}*)
Create a new array to hold the data based on the current set of fields, and store them in the *array* and member variable. Any data in the existing array will be lost.

nrows, if provided, is the number of rows to allocate.

`get_group_by_id`(*ref*, *before=None*)
Looks up a GROUP element by the given ID. Used by the group's "ref" attribute

`is_empty`()
Returns True if this table doesn't contain any real data because it was skipped over by the parser (through use of the `table_number` kwarg).

`to_xml`(*w*, ***kwargs*)

`iter_fields_and_params`()
Recursively iterate over all FIELD and PARAM elements in the TABLE.

`get_field_by_id_or_name`(*ref*, *before=None*)
Looks up a FIELD or PARAM element by the given ID or name.

classmethod `from_table`(*votable*, *table*)
Create a `Table` instance from a given `astropy.table.Table` instance.

Resource

class `astropy.io.votable.tree.Resource`(*name=None*, *ID=None*, *utype=None*, *type='results'*, *id=None*, *config={}*, *pos=None*, ***kwargs*)

Bases: `astropy.io.votable.tree.Element`, `astropy.io.votable.tree._IDProperty`, `astropy.io.votable.tree._NameProperty`, `astropy.io.votable.tree._UtypeProperty`, `astropy.io.votable.tree._DescriptionProperty`

RESOURCE element: Groups TABLE and RESOURCE elements.

The keyword arguments correspond to setting members of the same name, documented below.

Attributes Summary

`coordinate_systems`	A list of coordinate system definitions (COOSYS elements) for the RESOURCE.
`links`	A list of links (pointers to other documents or servers through a URI) for the resource.
`extra_attributes`	A dictionary of string keys to string values containing any extra attributes of the RESOURCE element that ar
`infos`	A list of informational parameters (key-value pairs) for the resource.
`tables`	A list of tables in the resource.
`params`	A list of parameters (constant-valued columns) for the resource.
`type`	[*required*] The type of the resource. Must be either:
`resources`	A list of nested resources inside this resource.

Methods Summary

parse(votable, iterator, config)	
to_xml(w, **kwargs)	
iter_fields_and_params()	Recursively iterates over all FIELD and PARAM elements in the resource, its tables and nested reso
iter_tables()	Recursively iterates over all tables in the resource and nested resources.
iter_coosys()	Recursively iterates over all the COOSYS elements in the resource and nested resources.

Attributes Documentation

`coordinate_systems`
A list of coordinate system definitions (COOSYS elements) for the RESOURCE. Must contain only `CooSys` objects.

`links`
A list of links (pointers to other documents or servers through a URI) for the resource. Must contain only `Link` objects.

`extra_attributes`
A dictionary of string keys to string values containing any extra attributes of the RESOURCE element that are not defined in the specification. (The specification explicitly allows for extra attributes here, but nowhere else.)

`infos`
A list of informational parameters (key-value pairs) for the resource. Must only contain `Info` objects.

`tables`
A list of tables in the resource. Must contain only `Table` objects.

`params`
A list of parameters (constant-valued columns) for the resource. Must contain only `Param` objects.

`type`
[*required*] The type of the resource. Must be either:

- 'results': This resource contains actual result values (default)
- 'meta': This resource contains only datatype descriptions (FIELD elements), but no actual data.

`resources`
A list of nested resources inside this resource. Must contain only `Resource` objects.

Methods Documentation

`parse(`*votable*, *iterator*, *config*`)`

`to_xml(`*w*, ***kwargs*`)`

`iter_fields_and_params()`
Recursively iterates over all FIELD and PARAM elements in the resource, its tables and nested resources.

`iter_tables()`
Recursively iterates over all tables in the resource and nested resources.

`iter_coosys()`
Recursively iterates over all the COOSYS elements in the resource and nested resources.

VOTableFile

class astropy.io.votable.tree.VOTableFile(*ID=None*, *id=None*, *config={}*, *pos=None*, *version='1.2'*)

Bases: astropy.io.votable.tree.Element, astropy.io.votable.tree._IDProperty, astropy.io.votable.tree._DescriptionProperty

VOTABLE element: represents an entire file.

The keyword arguments correspond to setting members of the same name, documented below.

version is settable at construction time only, since conformance tests for building the rest of the structure depend on it.

Attributes Summary

coordinate_systems	A list of coordinate system descriptions for the file.
groups	A list of groups, in the order they appear in the file.
infos	A list of informational parameters (key-value pairs) for the entire file.
version	The version of the VOTable specification that the file uses.
params	A list of parameters (constant-valued columns) that apply to the entire file.
resources	A list of resources, in the order they appear in the file.

Methods Summary

get_first_table()	Often, you know there is only one table in the file, and that's all you need.
get_coosys_by_id(ref[, before])	Looks up a COOSYS element by the given ID.
parse(iterator, config)	
get_field_by_id_or_name(ref[, before])	Looks up a FIELD element by the given ID or name.
get_table_by_id(ref[, before])	Looks up a TABLE element by the given ID.
to_xml(fd[, write_null_values, compressed, ...])	Write to an XML file.
iter_fields_and_params()	Recursively iterate over all FIELD and PARAM elements in the VOTABLE file.
iter_values()	Recursively iterate over all VALUES elements in the VOTABLE file.
get_values_by_id(ref[, before])	Looks up a VALUES element by the given ID.
set_all_tables_format(format)	Set the output storage format of all tables in the file.
iter_tables()	Iterates over all tables in the VOTable file in a "flat" way, ignoring the nesting of re
iter_coosys()	Recursively iterate over all COOSYS elements in the VOTABLE file.
get_field_by_id(ref[, before])	Looks up a FIELD element by the given ID.
iter_groups()	Recursively iterate over all GROUP elements in the VOTABLE file.
from_table(table[, table_id])	Create a VOTableFile instance from a given astropy.table.Table instance.
get_table_by_index(idx)	Get a table by its ordinal position in the file.
get_group_by_id(ref[, before])	Looks up a GROUP element by the given ID.

Attributes Documentation

coordinate_systems
: A list of coordinate system descriptions for the file. Must contain only CooSys objects.

groups
: A list of groups, in the order they appear in the file. Only supported as a child of the VOTABLE element in VOTable 1.2 or later.

infos
: A list of informational parameters (key-value pairs) for the entire file. Must only contain Info objects.

`version`
The version of the VOTable specification that the file uses.

`params`
A list of parameters (constant-valued columns) that apply to the entire file. Must contain only `Param` objects.

`resources`
A list of resources, in the order they appear in the file. Must only contain `Resource` objects.

Methods Documentation

`get_first_table()`
Often, you know there is only one table in the file, and that's all you need. This method returns that first table.

`get_coosys_by_id(`*ref*, *before=None*`)`
Looks up a COOSYS element by the given ID.

`parse(`*iterator*, *config*`)`

`get_field_by_id_or_name(`*ref*, *before=None*`)`
Looks up a FIELD element by the given ID or name.

`get_table_by_id(`*ref*, *before=None*`)`
Looks up a TABLE element by the given ID. Used by the table "ref" attribute.

`to_xml(`*fd*, *write_null_values=False*, *compressed=False*, *_debug_python_based_parser=False*, *_astropy_version=None*`)`
Write to an XML file.

Parameters

fd : str path or writable file-like object

Where to write the file.

write_null_values : bool, optional

When `True`, write the 'null' value (specified in the null attribute of the VALUES element for each FIELD) for empty values. When False (default), simply write no value.

compressed : bool, optional

When `True`, write to a gzip-compressed file. (Default: `False`)

`iter_fields_and_params()`
Recursively iterate over all FIELD and PARAM elements in the VOTABLE file.

`iter_values()`
Recursively iterate over all VALUES elements in the VOTABLE file.

`get_values_by_id(`*ref*, *before=None*`)`
Looks up a VALUES element by the given ID. Used by the values "ref" attribute.

`set_all_tables_format(`*format*`)`
Set the output storage format of all tables in the file.

`iter_tables()`
Iterates over all tables in the VOTable file in a "flat" way, ignoring the nesting of resources etc.

`iter_coosys()`
Recursively iterate over all COOSYS elements in the VOTABLE file.

`get_field_by_id(ref, before=None)`
Looks up a FIELD element by the given ID. Used by the field's "ref" attribute.

`iter_groups()`
Recursively iterate over all GROUP elements in the VOTABLE file.

classmethod `from_table(table, table_id=None)`
Create a `VOTableFile` instance from a given `astropy.table.Table` instance.

Parameters
table_id : str, optional

Set the given ID attribute on the returned Table instance.

`get_table_by_index(idx)`
Get a table by its ordinal position in the file.

`get_group_by_id(ref, before=None)`
Looks up a GROUP element by the given ID. Used by the group's "ref" attribute

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 crea

get_converter
`astropy.io.votable.converters.get_converter(field, config={}, pos=None)`
Get an appropriate converter instance for a given field.

Parameters
field : astropy.io.votable.tree.Field

config : dict, optional

Parser configuration dictionary

pos : tuple

Position in the input XML file. Used for error messages.

Returns
converter : astropy.io.votable.converters.Converter

table_column_to_votable_datatype
`astropy.io.votable.converters.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.

This necessarily must perform some heuristics to determine the type of variable length arrays fields, since they are not directly supported by Numpy.

If the column has dtype of "object", it performs the following tests:

- If all elements are byte or unicode strings, it creates a variable-length byte or unicode field, respectively.
- If all elements are numpy arrays of the same dtype and with a consistent shape in all but the first dimension, it creates a variable length array of fixed sized arrays. If the dtypes match, but the shapes do not, a variable length array is created.

If the dtype of the input is not understood, it sets the data type to the most inclusive: a variable length unicodeChar array.

Parameters

column : `astropy.table.Column` instance

Returns

attributes : dict

A dict containing 'datatype' and 'arraysize' keys that can be set on a VOTable FIELD element.

Classes

`Converter`(field[, config, pos])	The base class for all converters.

Converter

class `astropy.io.votable.converters.Converter`(*field*, *config={}*, *pos=None*)

Bases: `object`

The base class for all converters. Each subclass handles converting a specific VOTABLE data type to/from the TABLEDATA and BINARY on-disk representations.

Parameters

field : `Field`

object describing the datatype

config : dict

The parser configuration dictionary

pos : tuple

The position in the XML file where the FIELD object was found. Used for error messages.

Methods Summary

`binparse`(read)	Reads some number of bytes from the BINARY format representation by calling the function *r*
`parse`(value[, config, pos])	Convert the string *value* from the TABLEDATA format into an object with the correct native in
`output`(value, mask)	Convert the object *value* (in the native in-memory datatype) to a unicode string suitable for seri
`parse_scalar`(value[, config, pos])	Parse a single scalar of the underlying type of the converter.
`binoutput`(value, mask)	Convert the object *value* in the native in-memory datatype to a string of bytes suitable for serial

Methods Documentation

`binparse(`*read*`)`

Reads some number of bytes from the BINARY format representation by calling the function *read*, and returns the native in-memory object representation for the datatype handled by *self*.

Parameters

read : function

A function that given a number of bytes, returns a byte string.

Returns

native : tuple (value, mask)

The value as a Numpy array or scalar, and *mask* is True if the value is missing.

`parse(`*value*, *config={}*, *pos=None*`)`

Convert the string *value* from the TABLEDATA format into an object with the correct native in-memory datatype and mask flag.

Parameters

value : str

value in TABLEDATA format

Returns

native : tuple (value, mask)

The value as a Numpy array or scalar, and *mask* is True if the value is missing.

`output(`*value*, *mask*`)`

Convert the object *value* (in the native in-memory datatype) to a unicode string suitable for serializing in the TABLEDATA format.

Parameters

value : native type corresponding to this converter

The value

mask : bool

If `True`, will return the string representation of a masked value.

Returns

tabledata_repr : unicode

`parse_scalar(`*value*, *config={}*, *pos=None*`)`

Parse a single scalar of the underlying type of the converter. For non-array converters, this is equivalent to parse. For array converters, this is used to parse a single element of the array.

Parameters

value : str

value in TABLEDATA format

Returns

native : tuple (value, mask)

The value as a Numpy array or scalar, and *mask* is True if the value is missing.

`binoutput(`*value*, *mask*`)`

Convert the object *value* in the native in-memory datatype to a string of bytes suitable for serialization in the BINARY format.

Parameters

value : native type corresponding to this converter

The value

mask : bool

If `True`, will return the string representation of a masked value.

Returns

bytes : byte string

The binary representation of the value, suitable for serialization in the BINARY format.

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.

parse_ucd

`astropy.io.votable.ucd.parse_ucd`(*ucd*, *check_controlled_vocabulary=False*, *has_colon=False*)

Parse the UCD into its component parts.

Parameters

ucd : str

The UCD string

check_controlled_vocabulary : bool, optional

If `True`, then each word in the UCD will be verified against the UCD1+ controlled vocabulary, (as required by the VOTable specification version 1.2), otherwise not.

has_colon : bool, optional

If `True`, the UCD may contain a colon (as defined in earlier versions of the standard).

Returns

parts : list

The result is a list of tuples of the form:

(*namespace*, *word*)

If no namespace was explicitly specified, *namespace* will be returned as `'ivoa'` (i.e., the default namespace).

Raises

ValueError : *ucd* is invalid

check_ucd

`astropy.io.votable.ucd.check_ucd`(*ucd*, *check_controlled_vocabulary=False*, *has_colon=False*)

Returns False if *ucd* is not a valid unified content descriptor.

Parameters

ucd : str

The UCD string

check_controlled_vocabulary : bool, optional

If `True`, then each word in the UCD will be verified against the UCD1+ controlled vocabulary, (as required by the VOTable specification version 1.2), otherwise not.

Returns

valid : bool

astropy.io.votable.util Module

Various utilities and cookbook-like things.

Functions

`convert_to_writable_filelike`(*args, **kwds)	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.
`is_callable`(o)	Returns `True` if `o` is callable.

convert_to_writable_filelike

`astropy.io.votable.util.convert_to_writable_filelike`(**args*, ***kwds*)

Returns a writable file-like object suitable for streaming output.

Parameters

fd : file path string or writable file-like object

May be:

- a file path, in which case it is opened, and the file object is returned.
- an object with a `write()` method, in which case that object.

compressed : bool, optional

If `True`, create a gzip-compressed file. (Default is `False`).

Returns

fd : writable file-like object

coerce_range_list_param

`astropy.io.votable.util.coerce_range_list_param`(*p*, *frames=None*, *numeric=True*)

Coerces and/or verifies the object *p* into a valid range-list-format parameter.

As defined in Section 8.7.2 of Simple Spectral Access Protocol.

Parameters

p : str or sequence

May be a string as passed verbatim to the service expecting a range-list, or a sequence. If a sequence, each item must be either:

- a numeric value

- a named value, such as, for example, 'J' for named spectrum (if the *numeric* kwarg is False)
- a 2-tuple indicating a range
- the last item my be a string indicating the frame of reference

frames : sequence of str, optional

A sequence of acceptable frame of reference keywords. If not provided, the default set in `set_reference_frames` will be used.

numeric : bool, optional

TODO

Returns

parts : tuple

The result is a tuple:

- a string suitable for passing to a service as a range-list argument
- an integer counting the number of elements

is_callable

`astropy.io.votable.util.is_callable(o)`

Returns `True` if `o` is callable.

astropy.io.votable.validator Module

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.

make_validation_report

`astropy.io.votable.validator.main.make_validation_report(`*urls=None*, *destdir='astropy.io.votable.validator.results'*, *multiprocess=True*, *stilts=None*`)`

Validates a large collection of web-accessible VOTable files.

Generates a report as a directory tree of HTML files.

Parameters

urls : list of strings, optional

If provided, is a list of HTTP urls to download VOTable files from. If not provided, a built-in set of ~22,000 urls compiled by HEASARC will be used.

destdir : path, optional

The directory to write the report to. By default, this is a directory called `astropy.io.votable.validator.results` in the current directory. If the directory does not exist, it will be created.

multiprocess : bool, optional

If True (default), perform validations in parallel using all of the cores on this machine.

stilts : path, optional

To perform validation with votlint from the the Java-based STILTS VOTable parser, in addition to astropy.io.votable, set this to the path of the stilts.jar file. java on the system shell path will be used to run it.

Notes

Downloads of each given URL will be performed only once and cached locally in *destdir*. To refresh the cache, remove *destdir* first.

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.

check_id

astropy.io.votable.xmlutil.check_id(*ID*, *name='ID'*, *config={}*, *pos=None*)

Raises a VOTableSpecError if *ID* is not a valid XML ID.

name is the name of the attribute being checked (used only for error messages).

fix_id

astropy.io.votable.xmlutil.fix_id(*ID*, *config={}*, *pos=None*)

Given an arbitrary string, create one that can be used as an xml id.

This is rather simplistic at the moment, since it just replaces non-valid characters with underscores.

check_token

astropy.io.votable.xmlutil.check_token(*token*, *attr_name*, *config={}*, *pos=None*)

Raises a ValueError if *token* is not a valid XML token.

As defined by XML Schema Part 2.

check_mime_content_type

astropy.io.votable.xmlutil.check_mime_content_type(*content_type*, *config={}*, *pos=None*)

Raises a VOTableSpecError if *content_type* is not a valid MIME content type.

As defined by RFC 2045 (syntactically, at least).

check_anyuri
astropy.io.votable.xmlutil.check_anyuri(*uri*, *config={}*, *pos=None*)
Raises a VOTableSpecError if *uri* is not a valid URI.

As defined in RFC 2396.

validate_schema
astropy.io.votable.xmlutil.validate_schema(*filename*, *version='1.2'*)
Validates the given file against the appropriate VOTable schema.

Parameters
filename : str

The path to the XML file to validate

version : str

The VOTABLE version to check, which must be a string "1.0", "1.1", or "1.2".

For version "1.0", it is checked against a DTD, since that version did not have an XML Schema.

Returns
returncode, stdout, stderr : int, str, str

Returns the returncode from xmllint and the stdout and stderr as strings

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. 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 `vo.table` supports this convention when not in *Pedantic mode*.

`vo.table` 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, `vo.table` disallows the colon.

VOTable 1.2 corrects this error in the specification.

References: 1.1, XML Names

W03: Implictly 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 vo.table 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 unicode object To avoid local-dependent number parsing differences, vo.table 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. vo.table accepts id and emits this warning when not in pedantic mode.

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 vo.table.

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 `vo.table` 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 *pedantic mode* is off, `vo.table` 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
```

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 *pedantic mode* is off, `vo.table` 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 *pedantic mode* is off, 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 *pedantic mode* is off, 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: vo.table is designed for VOTable version 1.1 and 1.2, but this file is x Unknown issues may arise using `vo.table` with VOTable files from a version other than 1.1 or 1.2.

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 vo.table The VO catalog database retrieved from the www is designed for a newer version of vo.table. This may cause problems or limited features performing service queries. Consider upgrading vo.table 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 "-". In non-pedantic mode, 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 vo.table 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
```

References: 1.1, 1.2

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. Empty cell 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.

Exceptions

Note: This is a list of many of the fatal exceptions emitted by vo.table when the file does not conform to spec. Other exceptions may be raised due to unforeseen cases or bugs in vo.table 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
```

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.

Exception utilities

`astropy.io.votable.exceptions.warn_or_raise(`*warning_class*, *exception_class=None*, *args=()*, *config={}*, *pos=None*, *stacklevel=1*`)`
Warn or raise an exception, depending on the pedantic setting.

`astropy.io.votable.exceptions.vo_raise(`*exception_class*, *args=()*, *config={}*, *pos=None*`)`
Raise an exception, with proper position information if available.

`astropy.io.votable.exceptions.vo_reraise(`*exc*, *config={}*, *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={}*, *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={}*, *pos=None*`)`
Bases: `exceptions.Warning`

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={}*, *pos=None*`)`
Bases: `astropy.io.votable.exceptions.VOWarning`, `exceptions.SyntaxWarning`

A change has been made to the input XML file.

class `astropy.io.votable.exceptions.VOTableSpecWarning(`*args*, *config={}*, *pos=None*`)`
Bases: `astropy.io.votable.exceptions.VOWarning`, `exceptions.SyntaxWarning`

The input XML file violates the spec, but there is an obvious workaround.

class `astropy.io.votable.exceptions.UnimplementedWarning(`*args*, *config={}*, *pos=None*`)`
Bases: `astropy.io.votable.exceptions.VOWarning`, `exceptions.SyntaxWarning`

A feature of the VOTABLE spec is not implemented.

class `astropy.io.votable.exceptions.IOWarning(`*args*, *config={}*, *pos=None*`)`
Bases: `astropy.io.votable.exceptions.VOWarning`, `exceptions.RuntimeWarning`

A network or IO error occurred, but was recovered using the cache.

class `astropy.io.votable.exceptions.VOTableSpecError(`*args*, *config={}*, *pos=None*`)`
Bases: `astropy.io.votable.exceptions.VOWarning`, `exceptions.ValueError`

The input XML file violates the spec and there is no good workaround.

1.13 Miscellaneous Input/Output (`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 *Reading and writing Table objects*). Routines that are intended to be used directly by users are listed in the `astropy.io.misc` section.

1.13.1 astropy.io.misc Module

This package contains miscellaneous utility functions for data input/output with astropy.

Functions

fnpickle(object, fileorname[, usecPickle, ...])	Pickle an object to a specified file.
fnunpickle(fileorname[, number, usecPickle])	Unpickle pickled objects from a specified file and return the contents.

fnpickle

astropy.io.misc.pickle_helpers.fnpickle(*object*, *fileorname*, *usecPickle=True*, *protocol=None*, *append=False*)

Pickle an object to a specified file.

Parameters

object :

The python object to pickle.

fileorname : str or file-like

The filename or file into which the object should be pickled. If a file object, it should have been opened in binary mode.

usecPickle : bool

If True (default), the cPickle module is to be used in place of pickle (cPickle is faster). This only applies for python 2.x.

protocol : int or None

Pickle protocol to use - see the pickle module for details on these options. If None, the most recent protocol will be used.

append : bool

If True, the object is appended to the end of the file, otherwise the file will be overwritten (if a file object is given instead of a file name, this has no effect).

fnunpickle

astropy.io.misc.pickle_helpers.fnunpickle(*fileorname*, *number=0*, *usecPickle=True*)

Unpickle pickled objects from a specified file and return the contents.

Parameters

fileorname : str or file-like

The file name or file from which to unpickle objects. If a file object, it should have been opened in binary mode.

number : int

If 0, a single object will be returned (the first in the file). If >0, this specifies the number of objects to be unpickled, and a list will be returned with exactly that many objects. If <0, all objects in the file will be unpickled and returned as a list.

usecPickle : bool

If True, the `cPickle` module is to be used in place of `pickle` (cPickle is faster). This only applies for python 2.x.

Returns

contents : obj or list

If `number` is 0, this is a individual object - the first one unpickled from the file. Otherwise, it is a list of objects unpickled from the file.

Raises

EOFError :

If `number` is >0 and there are fewer than `number` objects in the pickled file.

1.13.2 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 *Reading and writing Table objects* for more details.

Functions

`read_table_hdf5`(input[, path])	Read a Table object from an HDF5 file This requires h5py to be installed.
`write_table_hdf5`(table, output[, path, ...])	Write a Table object to an HDF5 file This requires h5py to be installed.

read_table_hdf5

`astropy.io.misc.hdf5.read_table_hdf5`(*input*, *path=None*)

Read a Table object from an HDF5 file

This requires h5py to be installed.

Parameters

input : str or `h5py.highlevel.File` or `h5py.highlevel.Group`

If a string, the filename to read the table from. If an h5py object, either the file or the group object to read the table from.

path : str

The path from which to read the table inside the HDF5 file. This should be relative to the input file or group.

write_table_hdf5

`astropy.io.misc.hdf5.write_table_hdf5`(*table*, *output*, *path=None*, *compression=False*, *append=False*, *overwrite=False*)

Write a Table object to an HDF5 file

This requires h5py to be installed.

Parameters

output : str or `h5py.highlevel.File` or `h5py.highlevel.Group`

If a string, the filename to write the table to. If an h5py object, either the file or the group object to write the table to.

compression : bool

Whether to compress the table inside the HDF5 file.

path : str

The path to which to write the table inside the HDF5 file. This should be relative to the input file or group.

append : bool

Whether to append the table to an existing HDF5 file.

overwrite : bool

Whether to overwrite any existing file without warning.

1.14 World Coordinate System (`astropy.wcs`)

1.14.1 Introduction

`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 is at its base a wrapper around Mark Calabretta's wcslib, but also adds support for the Simple Imaging Polynomial (SIP) convention and table lookup distortions as defined in WCS Paper IV. Each of these transformations can be used independently or together in a standard pipeline.

1.14.2 Getting Started

The basic workflow is as follows:

1. `from astropy import wcs`
2. Call the `WCS` constructor with an `astropy.io.fits` header and/or hdulist object.
3. 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.
4. Use one of the following transformation methods:
    - `all_pix2world`: Perform all three transformations from pixel to world coordinates.
    - `wcs_pix2world`: Perform just the core WCS transformation from pixel to world coordinates.
    - `wcs_world2pix`: Perform just the core WCS transformation from world to pixel coordinates.
    - `sip_pix2foc`: Convert from pixel to focal plane coordinates using the SIP polynomial coefficients.
    - `sip_foc2pix`: Convert from focal plane to pixel coordinates using the SIP polynomial coefficients.
    - `p4_pix2foc`: Convert from pixel to focal plane coordinates using the table lookup distortion method described in Paper IV.
    - `det2im`: Convert from detector coordinates to image coordinates. Commonly used for narrow column correction.

1.14.3 Using `astropy.wcs`

Loading WCS information from a FITS file

This example loads a FITS file (supplied on the commandline) and uses the WCS cards in its primary header to transform.

```
# Load the WCS information from a fits header, and use it
# to convert pixel coordinates to world coordinates.

from __future__ import division # confidence high

import numpy
from astropy import wcs
from astropy.io import fits
import sys

# Load the FITS hdulist using astropy.io.fits
hdulist = fits.open(sys.argv[-1])

# 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()

# Some pixel coordinates of interest.
pixcrd = numpy.array([[0,0],[24,38],[45,98]], numpy.float_)

# Convert pixel coordinates to world coordinates
# The second argument is "origin" -- in this case we're declaring we
# have 1-based (Fortran-like) coordinates.
world = w.wcs_pix2world(pixcrd, 1)
print world

# Convert the same coordinates back to pixel coordinates.
pixcrd2 = w.wcs_world2pix(world, 1)
print pixcrd2

# These should be the same as the original pixel coordinates, modulo
# some floating-point error.
assert numpy.max(numpy.abs(pixcrd - pixcrd2)) < 1e-6
```

Building a WCS structure programmatically

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.

from __future__ import division # confidence high
```

```
import numpy
from astropy import wcs
from astropy.io import fits
import sys

# 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 = numpy.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)])

# Print out all of the contents of the WCS object
w.wcs.print_contents()

# Some pixel coordinates of interest.
pixcrd = numpy.array([[0,0],[24,38],[45,98]], numpy.float_)

# Convert pixel coordinates to world coordinates
world = w.wcs_pix2world(pixcrd, 1)
print world

# Convert the same coordinates back to pixel coordinates.
pixcrd2 = w.wcs_world2pix(world, 1)
print pixcrd2

# These should be the same as the original pixel coordinates, modulo
# some floating-point error.
assert numpy.max(numpy.abs(pixcrd - pixcrd2)) < 1e-6

# 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)
hdu.writeto('test.fits')
```

1.14.4 Other information

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_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_RADECSYS: Accept RADECSYS. This appeared in early drafts of WCS Paper I+II and was subsequently replaced by RADESYSa. The construtor 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 favour 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 obvious 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 (the obvious analogy) 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 an obvious 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.

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.

1.14.5 See Also

- wcslib

1.14.6 Reference/API

astropy.wcs Module

Introduction

`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 is at its base a wrapper around Mark Calabretta's wcslib, but also adds support for the Simple Imaging Polynomial (SIP) convention and table lookup distortions as defined in WCS Paper IV. Each of these transformations can be used independently or together in a standard pipeline.

Functions

`UnitConverter`(*args, **kwargs)	Deprecated since version 0.2.
`find_all_wcs`(header[, relax, keysel])	Find all the WCS transformations in the given header.

UnitConverter

`astropy.wcs.UnitConverter`(**args*, ***kwargs*)

Deprecated since version 0.2: Use astropy.units instead. UnitConverter(have, want, translate_units='')

An object for converting from one system of units to another.

Use the returned object's `convert` method to convert values from *have* to *want*.

This function is permissive in accepting whitespace in all contexts in a units specification where it does not create ambiguity (e.g. not between a metric prefix and a basic unit string), including in strings like `"log (m ** 2)"` which is formally disallowed.

Note: Deprecated in Astropy 0.2

`UnitConverter` will be removed in a future version of astropy. The `astropy.units` package should be used instead.

Parameters

have : string

FITS unit string to convert from, with or without surrounding square brackets (for inline specifications); text following the closing bracket is ignored.

want : string

FITS unit string to convert to, with or without surrounding square brackets (for inline specifications); text following the closing bracket is ignored.

ctrl : string, optional

Do potentially unsafe translations of non-standard unit strings.

Although `"S"` is commonly used to represent seconds, its recognizes `"S"` formally as Siemens, however rarely that may be translation to `"s"` is potentially unsafe since the standard used. The same applies to `"H"` for hours (Henry), and `"D"` for days (Debye).

This string controls what to do in such cases, and is case-insensitive.

•If the string contains "s", translate "S" to "s".

•If the string contains "h", translate "H" to "h".

•If the string contains "d", translate "D" to "d".

Thus '' doesn't do any unsafe translations, whereas 'shd' does all of them.

Raises

ValueError :

Invalid numeric multiplier.

SyntaxError :

Dangling binary operator.

SyntaxError :

Invalid symbol in INITIAL context.

SyntaxError :

Function in invalid context.

SyntaxError :

Invalid symbol in EXPON context.

SyntaxError :

Unbalanced bracket.

SyntaxError :

Unbalanced parenthesis.

SyntaxError :

Consecutive binary operators.

SyntaxError :

Internal parser error.

SyntaxError :

Non-conformant unit specifications.

SyntaxError :

Non-conformant functions.

ValueError :

Potentially unsafe translation.

find_all_wcs

astropy.wcs.wcs.find_all_wcs(*header*, *relax=True*, *keysel=None*)

Find all the WCS transformations in the given header.

Parameters

header : string or astropy.io.fits header object.

relax : bool or int, optional

Degree of permissiveness:

•True (default): Admit all recognized informal extensions of the WCS standard.

- `False`: Recognize only FITS keywords defined by the published WCS standard.
- `int`: a bit field selecting specific extensions to accept. See *Header-reading relaxation constants* for details.

keysel : sequence of flags, optional

A list of flags used to select the keyword types considered by wcslib. When `None`, only the standard image header keywords are considered (and the underlying wcspih() C function is called). To use binary table image array or pixel list keywords, *keysel* must be set.

Each element in the list should be one of the following strings:

- 'image': Image header keywords
- 'binary': Binary table image array keywords
- 'pixel': Pixel list keywords

Keywords such as `EQUIna` or `RFRQna` that are common to binary table image arrays and pixel lists (including `WCSNna` and `TWCSna`) are selected by both 'binary' and 'pixel'.

Returns

wcses : list of `WCS` objects

Classes

`DistortionLookupTable`	Represents a single lookup table for a Paper IV distortion transformation.
`FITSFixedWarning`	The warning raised when the contents of the FITS header have been modified to be standa
`Sip`	The `Sip` class performs polynomial distortion correction using the SIP convention in both
`Tabprm`	A class to store the information related to tabular coordinates, i.e., coordinates that are defi
`WCS`([header, fobj, key, minerr, relax, ...])	WCS objects perform standard WCS transformations, and correct for SIP and Paper IV tab
`Wcsprm`	`Wcsprm` is a direct wrapper around wcslib. It

DistortionLookupTable

class `astropy.wcs.DistortionLookupTable`

Bases: `object`

Represents a single lookup table for a Paper IV distortion transformation.

Parameters

table : 2-dimensional array

The distortion lookup table.

crpix : 2-tuple

The distortion array reference pixel

crval : 2-tuple

The image array pixel coordinate

cdelt : 2-tuple

The grid step size

Attributes Summary

crpix	double array[naxis] Coordinate reference pixels (CRPIXja) for
cdelt	double array[naxis] Coordinate increments (CDELTia) for each
crval	double array[naxis] Coordinate reference values (CRVALia) for
data	float array The array data for the

Methods Summary

get_offset(x, y) -> (x, y)	Returns the offset as defined in the distortion lookup table.

Attributes Documentation

crpix
: double array[naxis] Coordinate reference pixels (CRPIXja) for each pixel axis.

cdelt
: double array[naxis] Coordinate increments (CDELTia) for each coord axis.

If a CDi_ja linear transformation matrix is present, a warning is raised and cdelt is ignored. The CDi_ja matrix may be deleted by:

```
del wcs.wcs.cd
```

An undefined value is represented by NaN.

crval
: double array[naxis] Coordinate reference values (CRVALia) for each coordinate axis.

data
: float array The array data for the DistortionLookupTable.

Methods Documentation

get_offset(*x*, *y*) -> (*x*, *y*)
: Returns the offset as defined in the distortion lookup table.

Returns

coordinate : coordinate pair

The offset from the distortion table for pixel point (*x*, *y*).

FITSFixedWarning

exception astropy.wcs.wcs.FITSFixedWarning
: The warning raised when the contents of the FITS header have been modified to be standards compliant.

Sip

class astropy.wcs.Sip
: Bases: object

The Sip class performs polynomial distortion correction using the SIP convention in both directions.

Parameters

a : double array[m+1][m+1]

The `A_i_j` polynomial for pixel to focal plane transformation. Its size must be (m + 1, m + 1) where m = `A_ORDER`.

b : double array[m+1][m+1]

The `B_i_j` polynomial for pixel to focal plane transformation. Its size must be (m + 1, m + 1) where m = `B_ORDER`.

ap : double array[m+1][m+1]

The `AP_i_j` polynomial for pixel to focal plane transformation. Its size must be (m + 1, m + 1) where m = `AP_ORDER`.

bp : double array[m+1][m+1]

The `BP_i_j` polynomial for pixel to focal plane transformation. Its size must be (m + 1, m + 1) where m = `BP_ORDER`.

crpix : double array[2]

The reference pixel.

Notes

Shupe, D. L., M. Moshir, J. Li, D. Makovoz and R. Narron. 2005. "The SIP Convention for Representing Distortion in FITS Image Headers." ADASS XIV.

Attributes Summary

a	`double array[a_order+1][a_order+1]` Focal plane transformation
b	`double array[b_order+1][b_order+1]` Pixel to focal plane
b_order	`int` (read-only) Order of the polynomial (`B_ORDER`).
bp_order	`int` (read-only) Order of the polynomial (`BP_ORDER`).
ap	`double array[ap_order+1][ap_order+1]` Focal plane to pixel
a_order	`int` (read-only) Order of the polynomial (`A_ORDER`).
bp	`double array[bp_order+1][bp_order+1]` Focal plane to pixel
ap_order	`int` (read-only) Order of the polynomial (`AP_ORDER`).
crpix	`double array[naxis]` Coordinate reference pixels (`CRPIXja`) for

Methods Summary

pix2foc	sip_pix2foc(*pixcrd, origin*) -> double array[ncoord][nelem]
foc2pix	sip_foc2pix(*foccrd, origin*) -> double array[ncoord][nelem]

Attributes Documentation

a

`double array[a_order+1][a_order+1]` Focal plane transformation matrix.

The SIP `A_i_j` matrix used for pixel to focal plane transformation.

Its values may be changed in place, but it may not be resized, without creating a new Sip object.

b
double array[b_order+1][b_order+1] Pixel to focal plane transformation matrix.

The SIP B_i_j matrix used for pixel to focal plane transformation. Its values may be changed in place, but it may not be resized, without creating a new Sip object.

b_order
int (read-only) Order of the polynomial (B_ORDER).

bp_order
int (read-only) Order of the polynomial (BP_ORDER).

ap
double array[ap_order+1][ap_order+1] Focal plane to pixel transformation matrix.

The SIP AP_i_j matrix used for focal plane to pixel transformation. Its values may be changed in place, but it may not be resized, without creating a new Sip object.

a_order
int (read-only) Order of the polynomial (A_ORDER).

bp
double array[bp_order+1][bp_order+1] Focal plane to pixel transformation matrix.

The SIP BP_i_j matrix used for focal plane to pixel transformation. Its values may be changed in place, but it may not be resized, without creating a new Sip object.

ap_order
int (read-only) Order of the polynomial (AP_ORDER).

crpix
double array[naxis] Coordinate reference pixels (CRPIXja) for each pixel axis.

Methods Documentation

pix2foc()
sip_pix2foc(*pixcrd, origin*) -> double array[ncoord][nelem]

Convert pixel coordinates to focal plane coordinates using the SIP polynomial distortion convention.

Parameters
pixcrd : double array[ncoord][nelem]

Array of pixel coordinates.

origin : int

Specifies the origin of pixel values. The Fortran and FITS standards use an origin of 1. Numpy and C use array indexing with origin at 0.

Returns
foccrd : double array[ncoord][nelem]

Returns an array of focal plane coordinates.

Raises
MemoryError :

Memory allocation failed.

ValueError :

Invalid coordinate transformation parameters.

`foc2pix()`

sip_foc2pix(*foccrd, origin*) -> double array[ncoord][nelem]

Convert focal plane coordinates to pixel coordinates using the SIP polynomial distortion convention.

Parameters

foccrd : double array[ncoord][nelem]

Array of focal plane coordinates.

origin : int

Specifies the origin of pixel values. The Fortran and FITS standards use an origin of 1. Numpy and C use array indexing with origin at 0.

Returns

pixcrd : double array[ncoord][nelem]

Returns an array of pixel coordinates.

Raises

MemoryError :

Memory allocation failed.

ValueError :

Invalid coordinate transformation parameters.

Tabprm

class `astropy.wcs.Tabprm`

Bases: `object`

A class to store the information related to tabular coordinates, i.e., coordinates that are defined via a lookup table.

This class can not be constructed directly from Python, but instead is returned from `tab`.

Attributes Summary

`map`	`int array[M]` Association between axes.
`p0`	`int array[M]` Interpolated indices into the coordinate array.
`sense`	`int array[M]` +1 if monotonically increasing, -1 if decreasing.
`nc`	`int` (read-only) Total number of coord vectors in the coord array.
`M`	`int` (read-only) Number of tabular coordinate axes.
`coord`	`double array[K_M]...[K_2][K_1][M]` The tabular coordinate array.
`delta`	`double array[M]` (read-only) Interpolated indices into the coord
`extrema`	`double array[K_M]...[K_2][2][M]` (read-only)
`K`	`int array[M]` (read-only) The lengths of the axes of the coordinate
`crval`	`double array[M]` Index values for the reference pixel for each of

Methods Summary

`print_contents`()	Print the contents of the `Tabprm` object to stdout.

Continued on next page

Table 1.182 – continued from previous page

set()	Allocates memory for work arrays.

Attributes Documentation

map
: int array[M] Association between axes.

  A vector of length M that defines the association between axis *m* in the *M*-dimensional coordinate array (1 <= *m* <= *M*) and the indices of the intermediate world coordinate and world coordinate arrays.

  When the intermediate and world coordinate arrays contain the full complement of coordinate elements in image-order, as will usually be the case, then map[m-1] == i-1 for axis *i* in the *N*-dimensional image (1 <= *i* <= *N*). In terms of the FITS keywords:

  ```
  map[PVi_3a - 1] == i - 1.
  ```

  However, a different association may result if the intermediate coordinates, for example, only contains a (relevant) subset of intermediate world coordinate elements. For example, if *M* == 1 for an image with *N* > 1, it is possible to fill the intermediate coordinates with the relevant coordinate element with nelem set to 1. In this case map[0] = 0 regardless of the value of *i*.

p0
: int array[M] Interpolated indices into the coordinate array.

  Vector of length M of interpolated indices into the coordinate array such that Upsilon_m, as defined in Paper III, is equal to (p0[m] + 1) + delta[m].

sense
: int array[M] +1 if monotonically increasing, -1 if decreasing.

  A vector of length M whose elements indicate whether the corresponding indexing vector is monotonically increasing (+1), or decreasing (-1).

nc
: int (read-only) Total number of coord vectors in the coord array.

  Total number of coordinate vectors in the coordinate array being the product K_1 * K_2 * ... * K_M.

M
: int (read-only) Number of tabular coordinate axes.

coord
: double array[K_M]...[K_2][K_1][M] The tabular coordinate array.

  Has the dimensions:

  ```
  (K_M, ... K_2, K_1, M)
  ```

  (see K) i.e. with the M dimension varying fastest so that the M elements of a coordinate vector are stored contiguously in memory.

delta
: double array[M] (read-only) Interpolated indices into the coord array.

  Array of interpolated indices into the coordinate array such that Upsilon_m, as defined in Paper III, is equal to (p0 [m] + 1) + delta[m].

extrema
: double array[K_M]...[K_2][2][M] (read-only)

An array recording the minimum and maximum value of each element of the coordinate vector in each row of the coordinate array, with the dimensions:

```
(K_M, ... K_2, 2, M)
```

(see K). The minimum is recorded in the first element of the compressed K_1 dimension, then the maximum. This array is used by the inverse table lookup function to speed up table searches.

K
: `int array[M]` (read-only) The lengths of the axes of the coordinate array.

An array of length `M` whose elements record the lengths of the axes of the coordinate array and of each indexing vector.

crval
: `double array[M]` Index values for the reference pixel for each of the tabular coord axes.

Methods Documentation

print_contents()
: Print the contents of the `Tabprm` object to stdout. Probably only useful for debugging purposes, and may be removed in the future.

To get a string of the contents, use `repr`.

set()
: Allocates memory for work arrays.

Also sets up the class according to information supplied within it.

Note that this routine need not be called directly; it will be invoked by functions that need it.

Raises

MemoryError :

Memory allocation failed.

InvalidTabularParameters :

Invalid tabular parameters.

WCS

class astropy.wcs.wcs.WCS(*header=None*, *fobj=None*, *key=' '*, *minerr=0.0*, *relax=True*, *naxis=None*, *keysel=None*, *colsel=None*, *fix=True*)

Bases: astropy.wcs.WCSBase

WCS objects perform standard WCS transformations, and correct for SIP and Paper IV table-lookup distortions, based on the WCS keywords and supplementary data read from a FITS file.

Parameters

header : astropy.io.fits header object, string, dict-like, or None, optional

If *header* is not provided or None, the object will be initialized to default values.

fobj : An astropy.io.fits file (hdulist) object, optional

It is needed when header keywords point to a Paper IV Lookup table distortion stored in a different extension.

key : string, optional

The name of a particular WCS transform to use. This may be either `' '` or `'A'-'Z'` and corresponds to the `"a"` part of the `CTYPEia` cards. *key* may only be provided if *header* is also provided.

minerr : float, optional

The minimum value a distortion correction must have in order to be applied. If the value of `CQERRja` is smaller than *minerr*, the corresponding distortion is not applied.

relax : bool or int, optional

Degree of permissiveness:

- `True` (default): Admit all recognized informal extensions of the WCS standard.
- `False`: Recognize only FITS keywords defined by the published WCS standard.
- `int`: a bit field selecting specific extensions to accept. See *Header-reading relaxation constants* for details.

naxis : int or sequence, optional

Extracts specific coordinate axes using `sub()`. If a header is provided, and *naxis* is not `None`, *naxis* will be passed to `sub()` in order to select specific axes from the header. See `sub()` for more details about this parameter.

keysel : sequence of flags, optional

A sequence of flags used to select the keyword types considered by wcslib. When `None`, only the standard image header keywords are considered (and the underlying wcspih() C function is called). To use binary table image array or pixel list keywords, *keysel* must be set.

Each element in the list should be one of the following strings:

- 'image': Image header keywords
- 'binary': Binary table image array keywords
- 'pixel': Pixel list keywords

Keywords such as `EQUIna` or `RFRQna` that are common to binary table image arrays and pixel lists (including `WCSNna` and `TWCSna`) are selected by both 'binary' and 'pixel'.

colsel : sequence of int, optional

A sequence of table column numbers used to restrict the WCS transformations considered to only those pertaining to the specified columns. If `None`, there is no restriction.

fix : bool, optional

When `True` (default), call `fix` on the resulting object to fix any non-standard uses in the header. `FITSFixedWarning` Warnings will be emitted if any changes were made.

Raises

MemoryError :

Memory allocation failed.

ValueError :

Invalid key.

KeyError :

Key not found in FITS header.

AssertionError :

Lookup table distortion present in the header but *fobj* was not provided.

Notes

1. astropy.wcs supports arbitrary *n* dimensions for the core WCS (the transformations handled by WCSLIB). However, the Paper IV lookup table and SIP distortions must be two dimensional. Therefore, if you try to create a WCS object where the core WCS has a different number of dimensions than 2 and that object also contains a Paper IV lookup table or SIP distortion, a `ValueError` exception will be raised. To avoid this, consider using the *naxis* kwarg to select two dimensions from the core WCS.

2. The number of coordinate axes in the transformation is not determined directly from the `NAXIS` keyword but instead from the highest of:

   - `NAXIS` keyword
   - `WCSAXESa` keyword
   - The highest axis number in any parameterized WCS keyword. The keyvalue, as well as the keyword, must be syntactically valid otherwise it will not be considered.

   If none of these keyword types is present, i.e. if the header only contains auxiliary WCS keywords for a particular coordinate representation, then no coordinate description is constructed for it.

   The number of axes, which is set as the `naxis` member, may differ for different coordinate representations of the same image.

3. When the header includes duplicate keywords, in most cases the last encountered is used.

Attributes Summary

`naxis1`	Deprecated since version 0.2.
`naxis2`	Deprecated since version 0.2.

Methods Summary

`get_naxis`(*args, **kwargs)	Deprecated since version 0.2.
`pix2foc`(*args)	Convert pixel coordinates to focal plane coordinates using the SIP polynomial distort
`calcFootprint`([header, undistort, axes])	Calculates the footprint of the image on the sky.
`wcs_sky2pix`(*args, **kwargs)	Deprecated since version 0.0.
`printwcs`()	Temporary function for internal use.
`deepcopy`()	Return a deep copy of the object.
`sip_foc2pix`(*args)	Convert focal plane coordinates to pixel coordinates using the SIP polynomial distort
`to_header_string`([relax])	Identical to `to_header`, but returns a string containing the header cards.
`sub`(axes)	Extracts the coordinate description for a subimage from a `WCS` object.
`footprint_to_file`([filename, color, width])	Writes out a ds9 style regions file.
`to_fits`([relax])	Generate an `astropy.io.fits.HDUList` object with all of the information stored in
`wcs_pix2world`(*args, **kwargs)	Transforms pixel coordinates to world coordinates by doing only the basic wcslib tra
`wcs_world2pix`(*args, **kwargs)	Transforms world coordinates to pixel coordinates, using only the basic wcslib WCS
`p4_pix2foc`(*args)	Convert pixel coordinates to focal plane coordinates using Paper IV table-lookup dis

Table 1.184 – continued from previous page

rotateCD(theta)	
wcs_pix2sky(*args, **kwargs)	Deprecated since version 0.0.
get_axis_types()	Similar to self.wcsprm.axis_types but provides the information in a more Python
det2im(*args)	Convert detector coordinates to image plane coordinates using Paper IV table-lookup
copy()	Return a shallow copy of the object.
all_pix2world(*args, **kwargs)	Transforms pixel coordinates to world coordinates.
to_header([relax])	Generate an astropy.io.fits.Header object with the basic WCS and SIP informa
sip_pix2foc(*args)	Convert pixel coordinates to focal plane coordinates using the SIP polynomial distort
all_pix2sky(*args, **kwargs)	Deprecated since version 0.0.

Attributes Documentation

naxis1
: Deprecated since version 0.2: The naxis1 attribute is deprecated and may be removed in a future version.

naxis2
: Deprecated since version 0.2: The naxis2 attribute is deprecated and may be removed in a future version.

Methods Documentation

get_naxis(**args*, ***kwargs*)
: Deprecated since version 0.2: This method should not be public

pix2foc(**args*)
: Convert pixel coordinates to focal plane coordinates using the SIP polynomial distortion convention and Paper IV table-lookup distortion correction.

Parameters

args : flexible

There are two accepted forms for the positional arguments:

- 2 arguments: An *N* x *2* array of coordinates, and an *origin*.
- more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

Returns

result : array

Returns the focal coordinates. If the input was a single array and origin, a single array is returned, otherwise a tuple of arrays is returned.

Raises

MemoryError :

Memory allocation failed.

ValueError :

Invalid coordinate transformation parameters.

calcFootprint(*header=None*, *undistort=True*, *axes=None*)
: Calculates the footprint of the image on the sky.

A footprint is defined as the positions of the corners of the image on the sky after all available distortions have been applied.

Parameters

header : astropy.io.fits header object, optional

undistort : bool, optional

If `True`, take SIP and distortion lookup table into account

axes : length 2 sequence ints, optional

If provided, use the given sequence as the shape of the image. Otherwise, use the `NAXIS1` and `NAXIS2` keywords from the header that was used to create this `WCS` object.

Returns

coord : (4, 2) array of (*x*, *y*) coordinates.

`wcs_sky2pix(*args, **kwargs)`

Deprecated since version 0.0: Use wcs_world2pix instead.

`printwcs()`

Temporary function for internal use.

`deepcopy()`

Return a deep copy of the object.

Convenience method so user doesn't have to import the `copy` stdlib module.

`sip_foc2pix(*args)`

Convert focal plane coordinates to pixel coordinates using the SIP polynomial distortion convention.

Paper IV table lookup distortion correction is not applied, even if that information existed in the FITS file that initialized this `WCS` object.

Parameters

args : flexible

There are two accepted forms for the positional arguments:

- 2 arguments: An *N* x *2* array of coordinates, and an *origin*.
- more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

Returns

result : array

Returns the pixel coordinates. If the input was a single array and origin, a single array is returned, otherwise a tuple of arrays is returned.

Raises

MemoryError :

Memory allocation failed.

ValueError :

Invalid coordinate transformation parameters.

`to_header_string(relax=False)`

Identical to `to_header`, but returns a string containing the header cards.

sub(*axes*)

Extracts the coordinate description for a subimage from a WCS object.

The world coordinate system of the subimage must be separable in the sense that the world coordinates at any point in the subimage must depend only on the pixel coordinates of the axes extracted. In practice, this means that the PCi_ja matrix of the original image must not contain non-zero off-diagonal terms that associate any of the subimage axes with any of the non-subimage axes.

sub can also add axes to a wcsprm object. The new axes will be created using the defaults set by the Wcsprm constructor which produce a simple, unnamed, linear axis with world coordinates equal to the pixel coordinate. These default values can be changed before invoking set.

Parameters

axes : int or a sequence.

- If an int, include the first *N* axes in their original order.
- If a sequence, may contain a combination of image axis numbers (1-relative) or special axis identifiers (see below). Order is significant; axes[0] is the axis number of the input image that corresponds to the first axis in the subimage, etc. Use an axis number of 0 to create a new axis using the defaults.
- If 0, [] or None, do a deep copy.

Coordinate axes types may be specified using either strings or special integer constants. The available types are:

- 'longitude' / WCSSUB_LONGITUDE: Celestial longitude
- 'latitude' / WCSSUB_LATITUDE: Celestial latitude
- 'cubeface' / WCSSUB_CUBEFACE: Quadcube CUBEFACE axis
- 'spectral' / WCSSUB_SPECTRAL: Spectral axis
- 'stokes' / WCSSUB_STOKES: Stokes axis
- 'celestial' / WCSSUB_CELESTIAL: An alias for the combination of 'longitude', 'latitude' and 'cubeface'.

Returns

new_wcs : WCS object

Raises

MemoryError :

Memory allocation failed.

InvalidSubimageSpecificationError :

Invalid subimage specification (no spectral axis).

NonseparableSubimageCoordinateSystem :

Non-separable subimage coordinate system.

Notes

Combinations of subimage axes of particular types may be extracted in the same order as they occur in the input image by combining the integer constants with the 'binary or' (|) operator. For example:

```
wcs.sub([WCSSUB_LONGITUDE | WCSSUB_LATITUDE | WCSSUB_SPECTRAL])
```

would extract the longitude, latitude, and spectral axes in the same order as the input image. If one of each were present, the resulting object would have three dimensions.

For convenience, `WCSSUB_CELESTIAL` is defined as the combination `WCSSUB_LONGITUDE | WCSSUB_LATITUDE | WCSSUB_CUBEFACE`.

The codes may also be negated to extract all but the types specified, for example:

```
wcs.sub([
  WCSSUB_LONGITUDE,
  WCSSUB_LATITUDE,
  WCSSUB_CUBEFACE,
  -(WCSSUB_SPECTRAL | WCSSUB_STOKES)])
```

The last of these specifies all axis types other than spectral or Stokes. Extraction is done in the order specified by `axes`, i.e. a longitude axis (if present) would be extracted first (via `axes[0]`) and not subsequently (via `axes[3]`). Likewise for the latitude and cubeface axes in this example.

The number of dimensions in the returned object may be less than or greater than the length of `axes`. However, it will never exceed the number of axes in the input image.

`footprint_to_file`(*filename=None*, *color='green'*, *width=2*)

Writes out a ds9 style regions file. It can be loaded directly by ds9.

Parameters

filename : string, optional

Output file name - default is `'footprint.reg'`

color : string, optional

Color to use when plotting the line.

width : int, optional

Width of the region line.

`to_fits`(*relax=False*)

Generate an `astropy.io.fits.HDUList` object with all of the information stored in this object. This should be logically identical to the input FITS file, but it will be normalized in a number of ways.

See `to_header` for some warnings about the output produced.

Parameters

relax : bool or int, optional

Degree of permissiveness:

- `False` (default): Write all extensions that are considered to be safe and recommended.
- `True`: Write all recognized informal extensions of the WCS standard.
- `int`: a bit field selecting specific extensions to write. See *Header-writing relaxation constants* for details.

Returns

hdulist : `astropy.io.fits.HDUList`

`wcs_pix2world`(**args*, ***kwargs*)

Transforms pixel coordinates to world coordinates by doing only the basic wcslib transformation.

No SIP or Paper IV table lookup distortion correction is applied. To perform distortion correction, see `all_pix2world`, `sip_pix2foc`, `p4_pix2foc`, or `pix2foc`.

Parameters

args : flexible

There are two accepted forms for the positional arguments:

- 2 arguments: An *N* x *naxis* array of coordinates, and an *origin*.
- more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

For a transformation that is not two-dimensional, the two-argument form must be used.

ra_dec_order : bool, optional

When `True` will ensure that world coordinates are always given and returned in as (*ra*, *dec*) pairs, regardless of the order of the axes specified by the in the `CTYPE` keywords. Default is `False`.

Returns

result : array

Returns the world coordinates, in degrees. If the input was a single array and origin, a single array is returned, otherwise a tuple of arrays is returned.

Raises

MemoryError :

Memory allocation failed.

SingularMatrixError :

Linear transformation matrix is singular.

InconsistentAxisTypesError :

Inconsistent or unrecognized coordinate axis types.

ValueError :

Invalid parameter value.

ValueError :

Invalid coordinate transformation parameters.

ValueError :

x- and y-coordinate arrays are not the same size.

InvalidTransformError :

Invalid coordinate transformation parameters.

InvalidTransformError :

Ill-conditioned coordinate transformation parameters.

Notes

The order of the axes for the result is determined by the `CTYPEia` keywords in the FITS header, therefore it may not always be of the form (*ra*, *dec*). The `lat`, `lng`, `lattyp` and `lngtyp` members can be used to determine the order of the axes.

wcs_world2pix(*args, **kwargs)

Transforms world coordinates to pixel coordinates, using only the basic wcslib WCS transformation. No SIP or Paper IV table lookup distortion is applied.

Parameters

args : flexible

There are two accepted forms for the positional arguments:

- 2 arguments: An *N* x *naxis* array of coordinates, and an *origin*.
- more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

For a transformation that is not two-dimensional, the two-argument form must be used.

ra_dec_order : bool, optional

When `True` will ensure that world coordinates are always given and returned in as (*ra*, *dec*) pairs, regardless of the order of the axes specified by the in the `CTYPE` keywords. Default is `False`.

Returns

result : array

Returns the pixel coordinates. If the input was a single array and origin, a single array is returned, otherwise a tuple of arrays is returned.

Raises

MemoryError :

Memory allocation failed.

SingularMatrixError :

Linear transformation matrix is singular.

InconsistentAxisTypesError :

Inconsistent or unrecognized coordinate axis types.

ValueError :

Invalid parameter value.

ValueError :

Invalid coordinate transformation parameters.

ValueError :

x- and y-coordinate arrays are not the same size.

InvalidTransformError :

Invalid coordinate transformation parameters.

InvalidTransformError :

Ill-conditioned coordinate transformation parameters.

Notes

The order of the axes for the input world array is determined by the CTYPEia keywords in the FITS header, therefore it may not always be of the form (*ra*, *dec*). The lat, lng, lattyp and lngtyp members can be used to determine the order of the axes.

p4_pix2foc(**args*)

Convert pixel coordinates to focal plane coordinates using Paper IV table-lookup distortion correction.

Parameters

args : flexible

There are two accepted forms for the positional arguments:

- 2 arguments: An *N* x *2* array of coordinates, and an *origin*.
- more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

Returns

result : array

Returns the focal coordinates. If the input was a single array and origin, a single array is returned, otherwise a tuple of arrays is returned.

Raises

MemoryError :

Memory allocation failed.

ValueError :

Invalid coordinate transformation parameters.

rotateCD(*theta*)

wcs_pix2sky(**args*, ***kwargs*)

Deprecated since version 0.0: Use wcs_pix2world instead.

get_axis_types()

Similar to self.wcsprm.axis_types but provides the information in a more Python-friendly format.

Returns

result : list of dicts

Returns a list of dictionaries, one for each axis, each containing attributes about the type of that axis.

Each dictionary has the following keys:

- 'coordinate_type':
  - None: Non-specific coordinate type.
  - 'stokes': Stokes coordinate.
  - 'celestial': Celestial coordinate (including CUBEFACE).
  - 'spectral': Spectral coordinate.
- 'scale':

- 'linear': Linear axis.
- 'quantized': Quantized axis (STOKES, CUBEFACE).
- 'non-linear celestial': Non-linear celestial axis.
- 'non-linear spectral': Non-linear spectral axis.
- 'logarithmic': Logarithmic axis.
- 'tabular': Tabular axis.

• 'group'

- Group number, e.g. lookup table number

• 'number'

- For celestial axes:
  - 0: Longitude coordinate.
  - 1: Latitude coordinate.
  - 2: CUBEFACE number.
- For lookup tables:
  - the axis number in a multidimensional table.

CTYPEia in "4-3" form with unrecognized algorithm code will generate an error.

det2im(**args*)

Convert detector coordinates to image plane coordinates using Paper IV table-lookup distortion correction.

Parameters

args : flexible

There are two accepted forms for the positional arguments:

• 2 arguments: An *N* x *2* array of coordinates, and an *origin*.

• more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

Returns

result : array

Returns the pixel coordinates. If the input was a single array and origin, a single array is returned, otherwise a tuple of arrays is returned.

Raises

MemoryError :

Memory allocation failed.

ValueError :

Invalid coordinate transformation parameters.

copy()

Return a shallow copy of the object.

Convenience method so user doesn't have to import the copy stdlib module.

all_pix2world(*args, **kwargs)

Transforms pixel coordinates to world coordinates.

Performs all of the following in order:

- Detector to image plane correction (optionally)
- SIP distortion correction (optionally)
- Paper IV table-lookup distortion correction (optionally)
- wcslib WCS transformation

Parameters

args : flexible

There are two accepted forms for the positional arguments:

- 2 arguments: An *N* x *naxis* array of coordinates, and an *origin*.
- more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

For a transformation that is not two-dimensional, the two-argument form must be used.

ra_dec_order : bool, optional

When `True` will ensure that world coordinates are always given and returned in as (*ra*, *dec*) pairs, regardless of the order of the axes specified by the in the `CTYPE` keywords. Default is `False`.

Returns

result : array

Returns the sky coordinates, in degrees. If the input was a single array and origin, a single array is returned, otherwise a tuple of arrays is returned.

Raises

MemoryError :

Memory allocation failed.

SingularMatrixError :

Linear transformation matrix is singular.

InconsistentAxisTypesError :

Inconsistent or unrecognized coordinate axis types.

ValueError :

Invalid parameter value.

ValueError :

Invalid coordinate transformation parameters.

ValueError :

x- and y-coordinate arrays are not the same size.

InvalidTransformError :

Invalid coordinate transformation parameters.

InvalidTransformError :

Ill-conditioned coordinate transformation parameters.

Notes

The order of the axes for the result is determined by the `CTYPEia` keywords in the FITS header, therefore it may not always be of the form (*ra*, *dec*). The `lat`, `lng`, `lattyp` and `lngtyp` members can be used to determine the order of the axes.

`to_header(`*relax=False*`)`

Generate an `astropy.io.fits.Header` object with the basic WCS and SIP information stored in this object. This should be logically identical to the input FITS file, but it will be normalized in a number of ways.

> **Warning:** This function does not write out Paper IV distortion information, since that requires multiple FITS header data units. To get a full representation of everything in this object, use `to_fits`.

Parameters

relax : bool or int, optional

Degree of permissiveness:

- `False` (default): Write all extensions that are considered to be safe and recommended.
- `True`: Write all recognized informal extensions of the WCS standard.
- `int`: a bit field selecting specific extensions to write. See *Header-writing relaxation constants* for details.

Returns

header : `astropy.io.fits.Header`

Notes

The output header will almost certainly differ from the input in a number of respects:

1. The output header only contains WCS-related keywords. In particular, it does not contain syntactically-required keywords such as `SIMPLE`, `NAXIS`, `BITPIX`, or `END`.
2. Deprecated (e.g. `CROTAn`) or non-standard usage will be translated to standard (this is partially dependent on whether `fix` was applied).
3. Quantities will be converted to the units used internally, basically SI with the addition of degrees.
4. Floating-point quantities may be given to a different decimal precision.
5. Elements of the `PCi_j` matrix will be written if and only if they differ from the unit matrix. Thus, if the matrix is unity then no elements will be written.
6. Additional keywords such as `WCSAXES`, `CUNITia`, `LONPOLEa` and `LATPOLEa` may appear.
7. The original keycomments will be lost, although `to_header` tries hard to write meaningful comments.
8. Keyword order may be changed.

sip_pix2foc(**args*)

Convert pixel coordinates to focal plane coordinates using the SIP polynomial distortion convention.

Paper IV table lookup distortion correction is not applied, even if that information existed in the FITS file that initialized this WCS object. To correct for that, use pix2foc or p4_pix2foc.

Parameters

args : flexible

There are two accepted forms for the positional arguments:

- 2 arguments: An *N* x *2* array of coordinates, and an *origin*.
- more than 2 arguments: An array for each axis, followed by an *origin*. These arrays must be broadcastable to one another.

Here, *origin* is the coordinate in the upper left corner of the image. In FITS and Fortran standards, this is 1. In Numpy and C standards this is 0.

Returns

result : array

Returns the focal coordinates. If the input was a single array and origin, a single array is returned, otherwise a tuple of arrays is returned.

Raises

MemoryError :

Memory allocation failed.

ValueError :

Invalid coordinate transformation parameters.

all_pix2sky(**args*, ***kwargs*)

Deprecated since version 0.0: Use all_pix2world instead.

Wcsprm

class astropy.wcs.Wcsprm

Bases: object

Wcsprm is a direct wrapper around wcslib. It provides access to the core WCS transformations that it supports.

The FITS header parsing enforces correct FITS "keyword = value" syntax with regard to the equals sign occurring in columns 9 and 10. However, it does recognize free-format character (NOST 100-2.0, Sect. 5.2.1), integer (Sect. 5.2.3), and floating-point values (Sect. 5.2.4) for all keywords.

Parameters

header : An astropy.io.fits.Header, string, or None.

If None, the object will be initialized to default values.

key : string, optional

The key referring to a particular WCS transform in the header. This may be either ' ' or 'A'-'Z' and corresponds to the "a" part of "CTYPEia". (*key* may only be provided if *header* is also provided.)

relax : bool or int, optional

Degree of permissiveness:

- False: Recognize only FITS keywords defined by the published WCS standard.

- `True`: Admit all recognized informal extensions of the WCS standard.
- `int`: a bit field selecting specific extensions to accept. See *Header-reading relaxation constants* for details.

naxis : int, optional

The number of world coordinates axes for the object. (*naxis* may only be provided if *header* is `None`.)

keysel : sequence of flag bits, optional

Vector of flag bits that may be used to restrict the keyword types considered:

- `WCSHDR_IMGHEAD`: Image header keywords.
- `WCSHDR_BIMGARR`: Binary table image array.
- `WCSHDR_PIXLIST`: Pixel list keywords.

If zero, there is no restriction. If -1, the underlying wcslib function `wcspih()` is called, rather than `wcstbh()`.

colsel : sequence of int

A sequence of table column numbers used to restrict the keywords considered. `None` indicates no restriction.

Raises

MemoryError :

Memory allocation failed.

ValueError :

Invalid key.

KeyError :

Key not found in FITS header.

Attributes Summary

`theta0`	`double` The native longitude of the fiducial point.
`zsource`	`double` The redshift, `ZSOURCEa`, of the source.
`lonpole`	`double` The native longitude of the celestial pole.
`cd`	`double array[naxis][naxis]` The `CDi_ja` linear transformation
`tab`	`list of Tabprm` Tabular coordinate objects.
`cel_offset`	`boolean` Is there an offset?
`alt`	`str` Character code for alternate coordinate descriptions.
`restwav`	`double` Rest wavelength (m) from `RESTWAVa`.
`naxis`	`int` (read-only) The number of axes (pixel and coordinate).
`equinox`	`double` The equinox associated with dynamical equatorial or
`dateavg`	`string` Representative mid-point of the date of observation.
`pc`	`double array[naxis][naxis]` The `PCi_ja` (pixel coordinate)
`cname`	`list of strings` A list of the coordinate axis names, from
`restfrq`	`double` Rest frequency (Hz) from `RESTFRQa`.
`lattyp`	`string` (read-only) Celestial axis type for latitude.
`lngtyp`	`string` (read-only) Celestial axis type for longitude.

Continued on next page

Table 1.185 – continued from previous page

cubeface	int Index into the pixcrd (pixel coordinate) array for the
imgpix_matrix	double array[2][2] (read-only) Inverse of the CDELT or PC
obsgeo	double array[3] Location of the observer in a standard terrestrial
velangl	double Velocity angle.
name	string The name given to the coordinate representation
phi0	double The native latitude of the fiducial point.
dateobs	string Start of the date of observation.
crota	double array[naxis] CROTAia keyvalues for each coordinate
cdelt	double array[naxis] Coordinate increments (CDELTia) for each
piximg_matrix	double array[2][2] (read-only) Matrix containing the product of
ssysobs	string Spectral reference frame.
crpix	double array[naxis] Coordinate reference pixels (CRPIXja) for
crval	double array[naxis] Coordinate reference values (CRVALia) for
lat	int (read-only) The index into the world coord array containing
cunit	list of astropy.UnitBase[naxis] List of CUNITia keyvalues as
mjdavg	double Modified Julian Date corresponding to DATE-AVG.
mjdobs	double Modified Julian Date corresponding to DATE-OBS.
specsys	string Spectral reference frame (standard of rest), SPECSYSa.
colax	int array[naxis] An array recording the column numbers for each
spec	int (read-only) The index containing the spectral axis values.
colnum	int Column of FITS binary table associated with this WCS.
ssyssrc	string Spectral reference frame for redshift.
axis_types	int array[naxis] An array of four-digit type codes for each axis.
ctype	list of strings[naxis] List of CTYPEia keyvalues.
lng	int (read-only) The index into the world coord array containing
csyer	double array[naxis] The systematic error in the coordinate value
radesys	string The equatorial or ecliptic coordinate system type,
latpole	double The native latitude of the celestial pole, LATPOLEa (deg).
crder	double array[naxis] The random error in each coordinate axis,
velosys	double Relative radial velocity.

Methods Summary

cdfix()	Fix erroneously omitted CDi_ja keywords.
set()	Sets up a WCS object for use according to information supplied within it.
is_unity(() -> bool)	Returns True if the linear transformation matrix (cd) is unity.
unitfix([translate_units])	Translates non-standard CUNITia keyvalues.
cylfix()	Fixes WCS keyvalues for malformed cylindrical projections.
spcfix(() -> int)	Translates AIPS-convention spectral coordinate types.
sptr(ctype[, i])	Translates the spectral axis in a WCS object.
has_pci_ja(() -> bool)	Alias for has_pc.
fix([translate_units, naxis])	Applies all of the corrections handled separately by datfix, unitfix, celfix, spcfix,
get_pv(() -> list of tuples)	Returns PVi_ma keywords for each *i* and *m*.
set_ps(list)	Sets PSi_ma keywords for each *i* and *m*.
get_ps(() -> list of tuples)	Returns PSi_ma keywords for each *i* and *m*.
has_crotaia(() -> bool)	Alias for has_crota.
has_cd(() -> bool)	Returns True if CDi_ja is present.
mix(mixpix, mixcel, vspan, vstep, viter, ...)	Given either the celestial longitude or latitude plus an element of the pixel coordinate, so
s2p(world, origin)	Transforms world coordinates to pixel coordinates.

Table 1.186 – continued from previous

set_pv(list)	Sets PVi_ma keywords for each *i* and *m*.
get_pc(() -> double array[naxis][naxis])	Returns the PC matrix in read-only form.
print_contents()	Print the contents of the Wcsprm object to stdout.
celfix	Translates AIPS-convention celestial projection types, -NCP and -GLS.
get_cdelt(() -> double array[naxis])	Coordinate increments (CDELTia) for each coord axis.
p2s(pixcrd, origin)	Converts pixel to world coordinates.
has_pc(() -> bool)	Returns True if PCi_ja is present.
has_cdi_ja(() -> bool)	Alias for has_cd.
sub(axes)	Extracts the coordinate description for a subimage from a WCS object.
to_header([relax])	to_header translates a WCS object into a FITS header.
datfix()	Translates the old DATE-OBS date format to year-2000 standard form (yyyy-mm-ddThh:r
has_crota(() -> bool)	Returns True if CROTAia is present.

Attributes Documentation

theta0
: double The native longitude of the fiducial point.

  The point whose celestial coordinates are given in ref[1:2]. If undefined (NaN) the initialization routine, set, will set this to a projection-specific default.

  See Also:

  astropy.wcs.Wcsprm.phi0

zsource
: double The redshift, ZSOURCEa, of the source.

  An undefined value is represented by NaN.

lonpole
: double The native longitude of the celestial pole.

  LONPOLEa (deg).

cd
: double array[naxis][naxis] The CDi_ja linear transformation matrix.

  For historical compatibility, three alternate specifications of the linear transforations are available in wcslib. The canonical PCi_ja with CDELTia, and the deprecated CDi_ja and CROTAia keywords. Although the deprecated versions may not formally co-exist with PCi_ja, the approach here is simply to ignore them if given in conjunction with PCi_ja.

  has_pc, has_cd and has_crota can be used to determine which of these alternatives are present in the header.

  These alternate specifications of the linear transformation matrix are translated immediately to PCi_ja by set and are nowhere visible to the lower-level routines. In particular, set resets cdelt to unity if CDi_ja is present (and no PCi_ja). If no CROTAia is associated with the latitude axis, set reverts to a unity PCi_ja matrix.

tab
: list of Tabprm Tabular coordinate objects.

  A list of tabular coordinate objects associated with this WCS.

cel_offset
: boolean Is there an offset?

If True, an offset will be applied to (x, y) to force (x, y) = (0, 0) at the fiducial point, (phi_0, theta_0). Default is False.

alt
: str Character code for alternate coordinate descriptions.

  For example, the "a" in keyword names such as CTYPEia. This is a space character for the primary coordinate description, or one of the 26 upper-case letters, A-Z.

restwav
: double Rest wavelength (m) from RESTWAVa.

  An undefined value is represented by NaN.

naxis
: int (read-only) The number of axes (pixel and coordinate).

  Given by the NAXIS or WCSAXESa keyvalues.

  The number of coordinate axes is determined at parsing time, and can not be subsequently changed.

  It is determined from the highest of the following:

  1. NAXIS
  2. WCSAXESa
  3. The highest axis number in any parameterized WCS keyword. The keyvalue, as well as the keyword, must be syntactically valid otherwise it will not be considered.

  If none of these keyword types is present, i.e. if the header only contains auxiliary WCS keywords for a particular coordinate representation, then no coordinate description is constructed for it.

  This value may differ for different coordinate representations of the same image.

equinox
: double The equinox associated with dynamical equatorial or ecliptic coordinate systems.

  EQUINOXa (or EPOCH in older headers). Not applicable to ICRS equatorial or ecliptic coordinates.

  An undefined value is represented by NaN.

dateavg
: string Representative mid-point of the date of observation.

  In ISO format, yyyy-mm-ddThh:mm:ss.

  See Also:

  astropy.wcs.Wcsprm.dateobs

pc
: double array[naxis][naxis] The PCi_ja (pixel coordinate) transformation matrix.

  The order is:

```
[[PC1_1, PC1_2],
 [PC2_1, PC2_2]]
```

  For historical compatibility, three alternate specifications of the linear transforations are available in wcslib. The canonical PCi_ja with CDELTia, and the deprecated CDi_ja and CROTAia keywords. Although the deprecated versions may not formally co-exist with PCi_ja, the approach here is simply to ignore them if given in conjunction with PCi_ja.

  has_pc, has_cd and has_crota can be used to determine which of these alternatives are present in the header.

These alternate specifications of the linear transformation matrix are translated immediately to PCi_ja by set and are nowhere visible to the lower-level routines. In particular, set resets cdelt to unity if CDi_ja is present (and no PCi_ja). If no CROTAia is associated with the latitude axis, set reverts to a unity PCi_ja matrix.

cname
: list of strings A list of the coordinate axis names, from CNAMEia.

restfrq
: double Rest frequency (Hz) from RESTFRQa.

  An undefined value is represented by NaN.

lattyp
: string (read-only) Celestial axis type for latitude.

  For example, "RA", "DEC", "GLON", "GLAT", etc. extracted from "RA–", "DEC-", "GLON", "GLAT", etc. in the first four characters of CTYPEia but with trailing dashes removed.

lngtyp
: string (read-only) Celestial axis type for longitude.

  For example, "RA", "DEC", "GLON", "GLAT", etc. extracted from "RA–", "DEC-", "GLON", "GLAT", etc. in the first four characters of CTYPEia but with trailing dashes removed.

cubeface
: int Index into the pixcrd (pixel coordinate) array for the CUBEFACE axis.

  This is used for quadcube projections where the cube faces are stored on a separate axis.

  The quadcube projections (TSC, CSC, QSC) may be represented in FITS in either of two ways:

  - The six faces may be laid out in one plane and numbered as follows:

```
          0

4  3  2  1  4  3  2

          5
```

    Faces 2, 3 and 4 may appear on one side or the other (or both). The world-to-pixel routines map faces 2, 3 and 4 to the left but the pixel-to-world routines accept them on either side.

  - The COBE convention in which the six faces are stored in a three-dimensional structure using a CUBEFACE axis indexed from 0 to 5 as above.

  These routines support both methods; set determines which is being used by the presence or absence of a CUBEFACE axis in ctype. p2s and s2p translate the CUBEFACE axis representation to the single plane representation understood by the lower-level projection routines.

imgpix_matrix
: double array[2][2] (read-only) Inverse of the CDELT or PC matrix.

  Inverse containing the product of the CDELTia diagonal matrix and the PCi_ja matrix.

obsgeo
: double array[3] Location of the observer in a standard terrestrial reference frame.

  OBSGEO-X, OBSGEO-Y, OBSGEO-Z (in meters).

  An undefined value is represented by NaN.

velangl
: double Velocity angle.

The angle in degrees that should be used to decompose an observed velocity into radial and transverse components.

An undefined value is represented by NaN.

`name`
: `string` The name given to the coordinate representation `WCSNAMEa`.

`phi0`
: `double` The native latitude of the fiducial point.

The point whose celestial coordinates are given in `ref[1:2]`. If undefined (NaN) the initialization routine, `set`, will set this to a projection-specific default.

See Also:

`astropy.wcs.Wcsprm.theta0`

`dateobs`
: `string` Start of the date of observation.

In ISO format, `yyyy-mm-ddThh:mm:ss`.

See Also:

`astropy.wcs.Wcsprm.dateavg`

`crota`
: `double array[naxis]` `CROTAia` keyvalues for each coordinate axis.

For historical compatibility, three alternate specifications of the linear transforations are available in wcslib. The canonical `PCi_ja` with `CDELTia`, and the deprecated `CDi_ja` and `CROTAia` keywords. Although the deprecated versions may not formally co-exist with `PCi_ja`, the approach here is simply to ignore them if given in conjunction with `PCi_ja`.

`has_pc`, `has_cd` and `has_crota` can be used to determine which of these alternatives are present in the header.

These alternate specifications of the linear transformation matrix are translated immediately to `PCi_ja` by `set` and are nowhere visible to the lower-level routines. In particular, `set` resets `cdelt` to unity if `CDi_ja` is present (and no `PCi_ja`). If no `CROTAia` is associated with the latitude axis, `set` reverts to a unity `PCi_ja` matrix.

`cdelt`
: `double array[naxis]` Coordinate increments (`CDELTia`) for each coord axis.

If a `CDi_ja` linear transformation matrix is present, a warning is raised and `cdelt` is ignored. The `CDi_ja` matrix may be deleted by:

```
del wcs.wcs.cd
```

An undefined value is represented by NaN.

`piximg_matrix`
: `double array[2][2]` (read-only) Matrix containing the product of the `CDELTia` diagonal matrix and the `PCi_ja` matrix.

`ssysobs`
: `string` Spectral reference frame.

The spectral reference frame in which there is no differential variation in the spectral coordinate across the field-of-view, `SSYSOBSa`.

See Also:

astropy.wcs.Wcsprm.specsys, astropy.wcs.Wcsprm.velosys

crpix
: double array[naxis] Coordinate reference pixels (CRPIXja) for each pixel axis.

crval
: double array[naxis] Coordinate reference values (CRVALia) for each coordinate axis.

lat
: int (read-only) The index into the world coord array containing latitude values.

cunit
: list of astropy.UnitBase[naxis] List of CUNITia keyvalues as astropy.units.UnitBase instances.

  These define the units of measurement of the CRVALia, CDELTia and CDi_ja keywords.

  As CUNITia is an optional header keyword, cunit may be left blank but otherwise is expected to contain a standard units specification as defined by WCS Paper I. unitfix is available to translate commonly used non-standard units specifications but this must be done as a separate step before invoking set.

  For celestial axes, if cunit is not blank, set uses wcsunits to parse it and scale cdelt, crval, and cd to decimal degrees. It then resets cunit to "deg".

  For spectral axes, if cunit is not blank, set uses wcsunits to parse it and scale cdelt, crval, and cd to SI units. It then resets cunit accordingly.

  set ignores cunit for other coordinate types; cunit may be used to label coordinate values.

mjdavg
: double Modified Julian Date corresponding to DATE-AVG.

  (MJD = JD - 2400000.5).

  An undefined value is represented by NaN.

  See Also:

  astropy.wcs.Wcsprm.mjdobs

mjdobs
: double Modified Julian Date corresponding to DATE-OBS.

  (MJD = JD - 2400000.5).

  An undefined value is represented by NaN.

  See Also:

  astropy.wcs.Wcsprm.mjdavg

specsys
: string Spectral reference frame (standard of rest), SPECSYSa.

  See Also:

  astropy.wcs.Wcsprm.ssysobs, astropy.wcs.Wcsprm.velosys

colax
: int array[naxis] An array recording the column numbers for each axis in a pixel list.

spec
: int (read-only) The index containing the spectral axis values.

colnum
: int Column of FITS binary table associated with this WCS.

  Where the coordinate representation is associated with an image-array column in a FITS binary table, this property may be used to record the relevant column number.

  It should be set to zero for an image header or pixel list.

ssyssrc
: string Spectral reference frame for redshift.

  The spectral reference frame (standard of rest) in which the redshift was measured, SSYSSRCa.

axis_types
: int array[naxis] An array of four-digit type codes for each axis.

  - First digit (i.e. 1000s):
    - 0: Non-specific coordinate type.
    - 1: Stokes coordinate.
    - 2: Celestial coordinate (including CUBEFACE).
    - 3: Spectral coordinate.
  - Second digit (i.e. 100s):
    - 0: Linear axis.
    - 1: Quantized axis (STOKES, CUBEFACE).
    - 2: Non-linear celestial axis.
    - 3: Non-linear spectral axis.
    - 4: Logarithmic axis.
    - 5: Tabular axis.
  - Third digit (i.e. 10s):
    - 0: Group number, e.g. lookup table number
  - The fourth digit is used as a qualifier depending on the axis type.
    - For celestial axes:
      - 0: Longitude coordinate.
      - 1: Latitude coordinate.
      - 2: CUBEFACE number.
    - For lookup tables: the axis number in a multidimensional table.

  CTYPEia in "4-3" form with unrecognized algorithm code will have its type set to -1 and generate an error.

ctype
: list of strings[naxis] List of CTYPEia keyvalues.

  The ctype keyword values must be in upper case and there must be zero or one pair of matched celestial axis types, and zero or one spectral axis.

lng
: int (read-only) The index into the world coord array containing longitude values.

csyer
: double array[naxis] The systematic error in the coordinate value axes, CSYERia.

  An undefined value is represented by NaN.

radesys
: string The equatorial or ecliptic coordinate system type, RADESYSa.

latpole
: double The native latitude of the celestial pole, LATPOLEa (deg).

crder
: double array[naxis] The random error in each coordinate axis, CRDERia.

  An undefined value is represented by NaN.

velosys
: double Relative radial velocity.

  The relative radial velocity (m/s) between the observer and the selected standard of rest in the direction of the celestial reference coordinate, VELOSYSa.

  An undefined value is represented by NaN.

  See Also:

  astropy.wcs.Wcsprm.specsys, astropy.wcs.Wcsprm.ssysobs

Methods Documentation

cdfix()
: Fix erroneously omitted CDi_ja keywords.

  Sets the diagonal element of the CDi_ja matrix to unity if all CDi_ja keywords associated with a given axis were omitted. According to Paper I, if any CDi_ja keywords at all are given in a FITS header then those not given default to zero. This results in a singular matrix with an intersecting row and column of zeros.

  Returns

  success : int

  Returns 0 for success; -1 if no change required.

set()
: Sets up a WCS object for use according to information supplied within it.

  Note that this routine need not be called directly; it will be invoked by p2s and s2p if necessary.

  Some attributes that are based on other attributes (such as lattyp on ctype) may not be correct until after set is called.

  set strips off trailing blanks in all string members.

  set recognizes the NCP projection and converts it to the equivalent SIN projection and it also recognizes GLS as a synonym for SFL. It does alias translation for the AIPS spectral types (FREQ-LSR, FELO-HEL, etc.) but without changing the input header keywords.

  Raises

  MemoryError :

  Memory allocation failed.

  SingularMatrixError :

Linear transformation matrix is singular.

InconsistentAxisTypesError :

Inconsistent or unrecognized coordinate axis types.

ValueError :

Invalid parameter value.

InvalidTransformError :

Invalid coordinate transformation parameters.

InvalidTransformError :

Ill-conditioned coordinate transformation parameters.

`is_unity()` → bool

Returns `True` if the linear transformation matrix (`cd`) is unity.

`unitfix(`*translate_units*`='')`

Translates non-standard `CUNITia` keyvalues.

For example, `DEG` -> `deg`, also stripping off unnecessary whitespace.

Parameters

translate_units : string, optional

Do potentially unsafe translations of non-standard unit strings.

Although `"S"` is commonly used to represent seconds, its recognizes `"S"` formally as Siemens, however rarely that may be translation to `"s"` is potentially unsafe since the standard used. The same applies to `"H"` for hours (Henry), and `"D"` for days (Debye).

This string controls what to do in such cases, and is case-insensitive.

- If the string contains `"s"`, translate `"S"` to `"s"`.
- If the string contains `"h"`, translate `"H"` to `"h"`.
- If the string contains `"d"`, translate `"D"` to `"d"`.

Thus `''` doesn't do any unsafe translations, whereas `'shd'` does all of them.

Returns

success : int

Returns `0` for success; `-1` if no change required.

`cylfix()`

Fixes WCS keyvalues for malformed cylindrical projections.

Returns

success : int

Returns `0` for success; `-1` if no change required.

`spcfix()` → int

Translates AIPS-convention spectral coordinate types. `{FREQ, VELO, FELO}-{OBS, HEL, LSR}` (e.g. `FREQ-LSR`, `VELO-OBS`, `FELO-HEL`)

Returns

success : int

Returns `0` for success; `-1` if no change required.

sptr(*ctype*, *i=-1*)
Translates the spectral axis in a WCS object.

For example, a FREQ axis may be translated into ZOPT-F2W and vice versa.

Parameters

ctype : string

Required spectral CTYPEia, maximum of 8 characters. The first four characters are required to be given and are never modified. The remaining four, the algorithm code, are completely determined by, and must be consistent with, the first four characters. Wildcarding may be used, i.e. if the final three characters are specified as "???", or if just the eighth character is specified as "?", the correct algorithm code will be substituted and returned.

i : int

Index of the spectral axis (0-relative). If i < 0 (or not provided), it will be set to the first spectral axis identified from the CTYPE keyvalues in the FITS header.

Raises

MemoryError :

Memory allocation failed.

SingularMatrixError :

Linear transformation matrix is singular.

InconsistentAxisTypesError :

Inconsistent or unrecognized coordinate axis types.

ValueError :

Invalid parameter value.

InvalidTransformError :

Invalid coordinate transformation parameters.

InvalidTransformError :

Ill-conditioned coordinate transformation parameters.

InvalidSubimageSpecificationError :

Invalid subimage specification (no spectral axis).

has_pci_ja() → bool
Alias for has_pc. Maintained for backward compatibility.

fix(*translate_units=''*, *naxis=0*)
Applies all of the corrections handled separately by datfix, unitfix, celfix, spcfix, cylfix and cdfix.

Parameters

translate_units : str

Do potentially unsafe translations of non-standard unit strings.

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).

This string controls what to do in such cases, and is case-insensitive.

- If the string contains "s", translate "S" to "s".
- If the string contains "h", translate "H" to "h".
- If the string contains "d", translate "D" to "d".

Thus '' doesn't do any unsafe translations, whereas 'shd' does all of them.

naxis : int array[naxis]

Image axis lengths. If this array is set to zero or None, then cylfix will not be invoked.

Returns

status : dict

Returns a dictionary containing the following keys, each referring to a status string for each of the sub-fix functions that were called:

- cdfix
- datfix
- unitfix
- celfix
- spcfix
- cylfix

get_pv() → list of tuples

Returns PVi_ma keywords for each *i* and *m*.

Returns

Returned as a list of tuples of the form (*i*, *m*, *value*): :

- *i*: int. Axis number, as in PVi_ma, (i.e. 1-relative)
- *m*: int. Parameter number, as in PVi_ma, (i.e. 0-relative)
- *value*: string. Parameter value.

See Also:

astropy.wcs.Wcsprm.set_pv
Set PVi_ma values

Notes

Note that, if they were not given, set resets the entries for PVi_1a, PVi_2a, PVi_3a, and PVi_4a for longitude axis *i* to match (phi_0, theta_0), the native longitude and latitude of the reference point given by LONPOLEa and LATPOLEa.

set_ps(*list*)

Sets PSi_ma keywords for each *i* and *m*.

Parameters

ps : sequence of tuples

The input must be a sequence of tuples of the form (*i*, *m*, *value*):

- *i*: int. Axis number, as in PSi_ma, (i.e. 1-relative)

•*m*: int. Parameter number, as in PSi_ma, (i.e. 0-relative)

•*value*: string. Parameter value.

See Also:

astropy.wcs.Wcsprm.get_ps

get_ps() → list of tuples

Returns PSi_ma keywords for each *i* and *m*.

Returns

ps : list of tuples

Returned as a list of tuples of the form (*i*, *m*, *value*):

•*i*: int. Axis number, as in PSi_ma, (i.e. 1-relative)

•*m*: int. Parameter number, as in PSi_ma, (i.e. 0-relative)

•*value*: string. Parameter value.

See Also:

astropy.wcs.Wcsprm.set_ps

Set PSi_ma values

has_crotaia() → bool

Alias for has_crota. Maintained for backward compatibility.

has_cd() → bool

Returns True if CDi_ja is present.

CDi_ja is an alternate specification of the linear transformation matrix, maintained for historical compatibility.

Matrix elements in the IRAF convention are equivalent to the product CDi_ja = CDELTia * PCi_ja, but the defaults differ from that of the PCi_ja matrix. If one or more CDi_ja keywords are present then all unspecified CDi_ja default to zero. If no CDi_ja (or CROTAia) keywords are present, then the header is assumed to be in PCi_ja form whether or not any PCi_ja keywords are present since this results in an interpretation of CDELTia consistent with the original FITS specification.

While CDi_ja may not formally co-exist with PCi_ja, it may co-exist with CDELTia and CROTAia which are to be ignored.

See Also:

astropy.wcs.Wcsprm.cd

Get the raw CDi_ja values.

mix(*mixpix*, *mixcel*, *vspan*, *vstep*, *viter*, *world*, *pixcrd*, *origin*)

Given either the celestial longitude or latitude plus an element of the pixel coordinate, solves for the remaining elements by iterating on the unknown celestial coordinate element using s2p.

Parameters

mixpix : int

Which element on the pixel coordinate is given.

mixcel : int

Which element of the celestial coordinate is given. If *mixcel* = 1, celestial longitude is given in `world[self.lng]`, latitude returned in `world[self.lat]`. If *mixcel* = 2, celestial latitude is given in `world[self.lat]`, longitude returned in `world[self.lng]`.

vspan : pair of floats

Solution interval for the celestial coordinate, in degrees. The ordering of the two limits is irrelevant. Longitude ranges may be specified with any convenient normalization, for example `(-120,+120)` is the same as `(240,480)`, except that the solution will be returned with the same normalization, i.e. lie within the interval specified.

vstep : float

Step size for solution search, in degrees. If `0`, a sensible, although perhaps non-optimal default will be used.

viter : int

If a solution is not found then the step size will be halved and the search recommenced. *viter* controls how many times the step size is halved. The allowed range is 5 - 10.

world : double array[naxis]

World coordinate elements. `world[self.lng]` and `world[self.lat]` are the celestial longitude and latitude, in degrees. Which is given and which returned depends on the value of *mixcel*. All other elements are given. The results will be written to this array in-place.

pixcrd : double array[naxis].

Pixel coordinates. The element indicated by *mixpix* is given and the remaining elements will be written in-place.

origin : int

Specifies the origin of pixel values. The Fortran and FITS standards use an origin of 1. Numpy and C use array indexing with origin at 0.

Returns

result : dict

Returns a dictionary with the following keys:

- *phi* (double array[naxis])
- *theta* (double array[naxis])
  - Longitude and latitude in the native coordinate system of the projection, in degrees.
- *imgcrd* (double array[naxis])
  - Image coordinate elements. `imgcrd[self.lng]` and `imgcrd[self.lat]` are the projected *x*- and *y*-coordinates, in decimal degrees.
- *world* (double array[naxis])
  - Another reference to the *world* argument passed in.

Raises

MemoryError :

Memory allocation failed.

SingularMatrixError :

Linear transformation matrix is singular.

InconsistentAxisTypesError :

Inconsistent or unrecognized coordinate axis types.

ValueError :

Invalid parameter value.

InvalidTransformError :

Invalid coordinate transformation parameters.

InvalidTransformError :

Ill-conditioned coordinate transformation parameters.

InvalidCoordinateError :

Invalid world coordinate.

NoSolutionError :

No solution found in the specified interval.

See Also:

`astropy.wcs.Wcsprm.lat`, `astropy.wcs.Wcsprm.lng`

Notes

Initially, the specified solution interval is checked to see if it's a "crossing" interval. If it isn't, a search is made for a crossing solution by iterating on the unknown celestial coordinate starting at the upper limit of the solution interval and decrementing by the specified step size. A crossing is indicated if the trial value of the pixel coordinate steps through the value specified. If a crossing interval is found then the solution is determined by a modified form of "regula falsi" division of the crossing interval. If no crossing interval was found within the specified solution interval then a search is made for a "non-crossing" solution as may arise from a point of tangency. The process is complicated by having to make allowance for the discontinuities that occur in all map projections.

Once one solution has been determined others may be found by subsequent invocations of `mix` with suitably restricted solution intervals.

Note the circumstance that arises when the solution point lies at a native pole of a projection in which the pole is represented as a finite curve, for example the zenithals and conics. In such cases two or more valid solutions may exist but `mix` only ever returns one.

Because of its generality, `mix` is very compute-intensive. For compute-limited applications, more efficient special-case solvers could be written for simple projections, for example non-oblique cylindrical projections.

`s2p`(*world*, *origin*)

Transforms world coordinates to pixel coordinates.

Parameters

world : double array[ncoord][nelem]

Array of world coordinates, in decimal degrees.

origin : int

Specifies the origin of pixel values. The Fortran and FITS standards use an origin of 1. Numpy and C use array indexing with origin at 0.

Returns

result : dict

Returns a dictionary with the following keys:

- *phi*: double array[ncoord]
- *theta*: double array[ncoord]
  - Longitude and latitude in the native coordinate system of the projection, in degrees.
- *imgcrd*: double array[ncoord][nelem]
  - Array of intermediate world coordinates. For celestial axes, `imgcrd[][self.lng]` and `imgcrd[][self.lat]` are the projected *x*-, and *y*-coordinates, in pseudo "degrees". For quad-cube projections with a `CUBEFACE` axis, the face number is also returned in `imgcrd[][self.cubeface]`. For spectral axes, `imgcrd[][self.spec]` is the intermediate spectral coordinate, in SI units.
- *pixcrd*: double array[ncoord][nelem]
  - Array of pixel coordinates. Pixel coordinates are zero-based.
- *stat*: int array[ncoord]
  - Status return value for each coordinate. `0` for success, `1+` for invalid pixel coordinate.

Raises

MemoryError :

Memory allocation failed.

SingularMatrixError :

Linear transformation matrix is singular.

InconsistentAxisTypesError :

Inconsistent or unrecognized coordinate axis types.

ValueError :

Invalid parameter value.

InvalidTransformError :

Invalid coordinate transformation parameters.

InvalidTransformError :

Ill-conditioned coordinate transformation parameters.

See Also:

`astropy.wcs.Wcsprm.lat`, `astropy.wcs.Wcsprm.lng`

`set_pv(`*list*`)`

Sets `PVi_ma` keywords for each *i* and *m*.

Parameters

pv : list of tuples

The input must be a sequence of tuples of the form (*i*, *m*, *value*):

- *i*: int. Axis number, as in `PVi_ma`, (i.e. 1-relative)
- *m*: int. Parameter number, as in `PVi_ma`, (i.e. 0-relative)
- *value*: float. Parameter value.

See Also:

`astropy.wcs.Wcsprm.get_pv`

`get_pc()` → double array[naxis][naxis]

Returns the `PC` matrix in read-only form. Unlike the `pc` property, this works even when the header specifies the linear transformation matrix in one of the deprecated `CDi_ja` or `CROTAia` forms. This is useful when you want access to the linear transformation matrix, but don't care how it was specified in the header.

`print_contents()`

Print the contents of the `Wcsprm` object to stdout. Probably only useful for debugging purposes, and may be removed in the future.

To get a string of the contents, use `repr`.

`celfix()`

Translates AIPS-convention celestial projection types, `-NCP` and `-GLS`.

Returns

success : int

Returns `0` for success; `-1` if no change required.

`get_cdelt()` → double array[naxis]

Coordinate increments (`CDELTia`) for each coord axis.

Returns the `CDELT` offsets in read-only form. Unlike the `cdelt` property, this works even when the header specifies the linear transformation matrix in one of the deprecated `CDi_ja` or `CROTAia` forms. This is useful when you want access to the linear transformation matrix, but don't care how it was specified in the header.

`p2s(`*pixcrd*, *origin*`)`

Converts pixel to world coordinates.

Parameters

pixcrd : double array[ncoord][nelem]

Array of pixel coordinates.

origin : int

Specifies the origin of pixel values. The Fortran and FITS standards use an origin of 1. Numpy and C use array indexing with origin at 0.

Returns

result : dict

Returns a dictionary with the following keys:

- *imgcrd*: double array[ncoord][nelem]
  - Array of intermediate world coordinates. For celestial axes, `imgcrd[][self.lng]` and `imgcrd[][self.lat]` are the projected

x-, and *y*-coordinates, in pseudo degrees. For spectral axes, `imgcrd[][self.spec]` is the intermediate spectral coordinate, in SI units.

- *phi*: double array[ncoord]
- *theta*: double array[ncoord]
  - Longitude and latitude in the native coordinate system of the projection, in degrees.
- *world*: double array[ncoord][nelem]
  - Array of world coordinates. For celestial axes, `world[][self.lng]` and `world[][self.lat]` are the celestial longitude and latitude, in degrees. For spectral axes, `world[][self.spec]` is the intermediate spectral coordinate, in SI units.
- *stat*: int array[ncoord]
  - Status return value for each coordinate. `0` for success, `1+` for invalid pixel coordinate.

Raises

MemoryError :

Memory allocation failed.

SingularMatrixError :

Linear transformation matrix is singular.

InconsistentAxisTypesError :

Inconsistent or unrecognized coordinate axis types.

ValueError :

Invalid parameter value.

ValueError :

x- and *y*-coordinate arrays are not the same size.

InvalidTransformError :

Invalid coordinate transformation parameters.

InvalidTransformError :

Ill-conditioned coordinate transformation parameters.

See Also:

`astropy.wcs.Wcsprm.lat`, `astropy.wcs.Wcsprm.lng`

`has_pc()` → bool

Returns `True` if `PCi_ja` is present. `PCi_ja` is the recommended way to specify the linear transformation matrix.

See Also:

`astropy.wcs.Wcsprm.pc`

Get the raw `PCi_ja` values

has_cdi_ja() → bool
Alias for has_cd. Maintained for backward compatibility.

sub(*axes*)
Extracts the coordinate description for a subimage from a WCS object.

The world coordinate system of the subimage must be separable in the sense that the world coordinates at any point in the subimage must depend only on the pixel coordinates of the axes extracted. In practice, this means that the PCi_ja matrix of the original image must not contain non-zero off-diagonal terms that associate any of the subimage axes with any of the non-subimage axes.

sub can also add axes to a wcsprm object. The new axes will be created using the defaults set by the Wcsprm constructor which produce a simple, unnamed, linear axis with world coordinates equal to the pixel coordinate. These default values can be changed before invoking set.

Parameters
axes : int or a sequence.

- If an int, include the first *N* axes in their original order.
- If a sequence, may contain a combination of image axis numbers (1-relative) or special axis identifiers (see below). Order is significant; axes[0] is the axis number of the input image that corresponds to the first axis in the subimage, etc. Use an axis number of 0 to create a new axis using the defaults.
- If 0, [] or None, do a deep copy.

Coordinate axes types may be specified using either strings or special integer constants. The available types are:

- 'longitude' / WCSSUB_LONGITUDE: Celestial longitude
- 'latitude' / WCSSUB_LATITUDE: Celestial latitude
- 'cubeface' / WCSSUB_CUBEFACE: Quadcube CUBEFACE axis
- 'spectral' / WCSSUB_SPECTRAL: Spectral axis
- 'stokes' / WCSSUB_STOKES: Stokes axis
- 'celestial' / WCSSUB_CELESTIAL: An alias for the combination of 'longitude', 'latitude' and 'cubeface'.

Returns
new_wcs : WCS object

Raises
MemoryError :

Memory allocation failed.

InvalidSubimageSpecificationError :

Invalid subimage specification (no spectral axis).

NonseparableSubimageCoordinateSystem :

Non-separable subimage coordinate system.

Notes

Combinations of subimage axes of particular types may be extracted in the same order as they occur in the input image by combining the integer constants with the 'binary or' (|) operator. For example:

```
wcs.sub([WCSSUB_LONGITUDE | WCSSUB_LATITUDE | WCSSUB_SPECTRAL])
```

would extract the longitude, latitude, and spectral axes in the same order as the input image. If one of each were present, the resulting object would have three dimensions.

For convenience, WCSSUB_CELESTIAL is defined as the combination WCSSUB_LONGITUDE | WCSSUB_LATITUDE | WCSSUB_CUBEFACE.

The codes may also be negated to extract all but the types specified, for example:

```
wcs.sub([
  WCSSUB_LONGITUDE,
  WCSSUB_LATITUDE,
  WCSSUB_CUBEFACE,
  -(WCSSUB_SPECTRAL | WCSSUB_STOKES)])
```

The last of these specifies all axis types other than spectral or Stokes. Extraction is done in the order specified by axes, i.e. a longitude axis (if present) would be extracted first (via axes[0]) and not subsequently (via axes[3]). Likewise for the latitude and cubeface axes in this example.

The number of dimensions in the returned object may be less than or greater than the length of axes. However, it will never exceed the number of axes in the input image.

to_header(*relax=False*)

to_header translates a WCS object into a FITS header.

The details of the header depends on context:

- If the colnum member is non-zero then a binary table image array header will be produced.
- Otherwise, if the colax member is set non-zero then a pixel list header will be produced.
- Otherwise, a primary image or image extension header will be produced.

The output header will almost certainly differ from the input in a number of respects:

1. The output header only contains WCS-related keywords. In particular, it does not contain syntactically-required keywords such as SIMPLE, NAXIS, BITPIX, or END.
2. Deprecated (e.g. CROTAn) or non-standard usage will be translated to standard (this is partially dependent on whether fix was applied).
3. Quantities will be converted to the units used internally, basically SI with the addition of degrees.
4. Floating-point quantities may be given to a different decimal precision.
5. Elements of the PCi_j matrix will be written if and only if they differ from the unit matrix. Thus, if the matrix is unity then no elements will be written.
6. Additional keywords such as WCSAXES, CUNITia, LONPOLEa and LATPOLEa may appear.
7. The original keycomments will be lost, although to_header tries hard to write meaningful comments.
8. Keyword order may be changed.

Keywords can be translated between the image array, binary table, and pixel lists forms by manipulating the colnum or colax members of the WCS object.

Parameters

relax : bool or int

Degree of permissiveness:

- `False`: Recognize only FITS keywords defined by the published WCS standard.
- `True`: Admit all recognized informal extensions of the WCS standard.
- `int`: a bit field selecting specific extensions to write. See *Header-writing relaxation constants* for details.

Returns

header : str

Raw FITS header as a string.

`datfix()`

Translates the old `DATE-OBS` date format to year-2000 standard form `(yyyy-mm-ddThh:mm:ss)` and derives `MJD-OBS` from it if not already set.

Alternatively, if `mjdobs` is set and `dateobs` isn't, then `datfix` derives `dateobs` from it. If both are set but disagree by more than half a day then `ValueError` is raised.

Returns

success : int

Returns `0` for success; `-1` if no change required.

`has_crota()` → bool

Returns `True` if `CROTAia` is present.

`CROTAia` is an alternate specification of the linear transformation matrix, maintained for historical compatibility.

In the AIPS convention, `CROTAia` may only be associated with the latitude axis of a celestial axis pair. It specifies a rotation in the image plane that is applied *after* the `CDELTia`; any other `CROTAia` keywords are ignored.

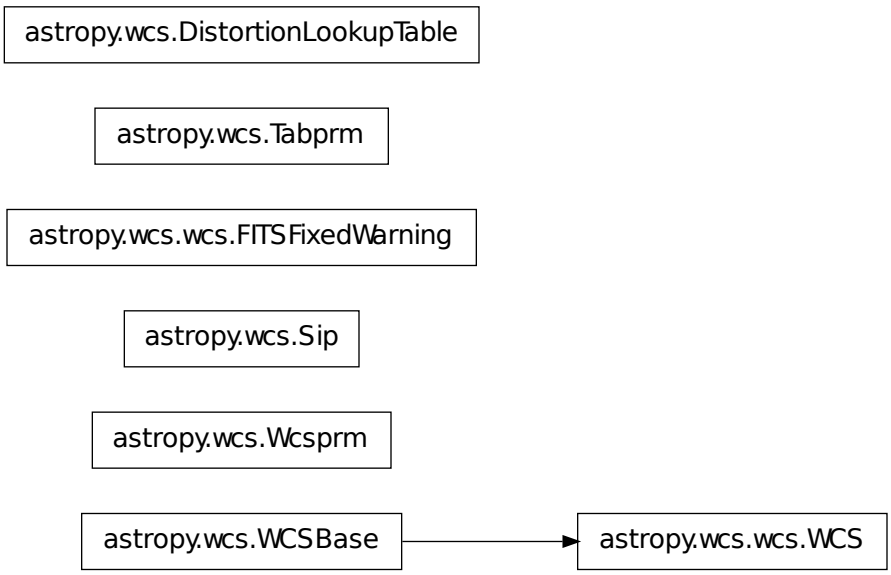
`CROTAia` may not formally co-exist with `PCi_ja`. `CROTAia` and `CDELTia` may formally co-exist with `CDi_ja` but if so are to be ignored.

See Also:

`astropy.wcs.Wcsprm.crota`

Get the raw `CROTAia` values

Class Inheritance Diagram



1.14.7 Acknowledgments and Licenses

wcslib is licenced under the GNU Lesser General Public License.

1.15 Astrostatistics Tools (`astropy.stats`)

1.15.1 Introduction

The `astropy.stats` package holds statsitical functions or algorithms used in astronomy and astropy.

1.15.2 Getting Started

The current tools are fairly self-contained, and include relevant examples in their docstrings. For example, see `sigma_clip`.

1.15.3 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.

1.15.4 Reference/API

astropy.stats Module

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

`sigma_clip`(data[, sig, iters, cenfunc, ...])	Perform sigma-clipping on the provided data.

sigma_clip

`astropy.stats.funcs.sigma_clip`(*data*, *sig=3*, *iters=1*, *cenfunc=<function median at 0x3823ed8>*, *varfunc=<function var at 0x3716f50>*, *maout=False*)

Perform sigma-clipping on the provided data.

This performs the sigma clipping algorithm - i.e. the data will be iterated over, each time rejecting points that are more than a specified number of standard deviations discrepant.

Note: `scipy.stats.sigmaclip` provides a subset of the functionality in this function.

Parameters

data : array-like

The data to be sigma-clipped (any shape).

sig : float

The number of standard deviations (*not* variances) to use as the clipping limit.

iters : int or None

The number of iterations to perform clipping for, or None to clip until convergence is achieved (i.e. continue until the last iteration clips nothing).

cenfunc : callable

The technique to compute the center for the clipping. Must be a callable that takes in a 1D data array and outputs the central value. Defaults to the median.

varfunc : callable

The technique to compute the variance about the center. Must be a callable that takes in a 1D data array and outputs the width estimator that will be interpreted as a variance. Defaults to the variance.

maout : bool or 'copy'

If True, a masked array will be returned. If the special string 'inplace', the masked array will contain the same array as `data`, otherwise the array data will be copied.

Returns

filtereddata : `numpy.ndarray` or `numpy.masked.MaskedArray`

If maout is True, this is a masked array with a shape matching the input that is masked where the algorithm has rejected those values. Otherwise, a 1D array of values including only those that are not clipped.

mask : boolean array

Only present if maout is False. A boolean array with a shape matching the input data that is False for rejected values and True for all others.

Examples

This will generate random variates from a Gaussian distribution and return only the points that are within 2 *sample* standard deviation from the median:

```
>>> from astropy.tools.alg import sigma_clip
>>> from numpy.random import randn
>>> randvar = randn(10000)
>>> data,mask = sigma_clip(randvar, 2, 1)
```

This will clipping on a similar distribution, but for 3 sigma relative to the sample *mean*, will clip until converged, and produces a numpy.masked.MaskedArray:

```
>>> from astropy.tools.alg import sigma_clip
>>> from numpy.random import randn
>>> from numpy import mean
>>> randvar = randn(10000)
>>> maskedarr = sigma_clip(randvar, 3, None, mean, maout=True)
```

1.16 Astropy Core Package Utilities (astropy.utils)

1.16.1 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 particuly 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 does not have detailed sections like the other packages.

Note: The astropy.utils.compat subpackage is not included in this documentation. It contains utility modules for compatibility with older/newer versions of python, 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 it cannot be reliably included here because of the large amount of version-specific code it contains.

1.16.2 Reference/API

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

find_current_module([depth, finddiff])	Determines the module/package from which this function is called.
isiterable(obj)	Returns True if the given object is iterable.
deprecated(since[, message, name, ...])	Used to mark a function as deprecated.
deprecated_attribute(name, since[, message, ...])	Used to mark a public attribute as deprecated.
format_exception(msg, *args, **kwargs)	Given an exception message string, uses new-style formatting arguments {fil

find_current_module

astropy.utils.misc.find_current_module(*depth=1*, *finddiff=False*)

Determines the module/package from which this function is called.

This function has two modes, determined by the finddiff option. it will either simply go the requested number of frames up the call stack (if finddiff is False), or it will go up the call stack until it reaches a module that is *not* in a specified set.

Parameters

depth : int

Specifies how far back to go in the call stack (0-indexed, so that passing in 0 gives back astropy.utils.misc).

finddiff : bool or list

If False, the returned mod will just be depth frames up from the current frame. Otherwise, the function will start at a frame depth up from current, and continue up the call stack to the first module that is *different* from those in the provided list. In this case, finddiff can be a list of modules or modules names. Alternatively, it can be True, which will use the module depth call stack frames up as the module the returned module most be different from.

Returns

mod : module or None

The module object or None if the package cannot be found. The name of the module is available as the __name__ attribute of the returned object (if it isn't None).

Raises

ValueError :

If finddiff is a list with an invalid entry.

Examples

The examples below assume that there are two modules in a package named pkg. mod1.py:

```
def find1():
    from astropy.utils import find_current_module
    print find_current_module(1).__name__
def find2():
    from astropy.utils import find_current_module
    cmod = find_current_module(2)
    if cmod is None:
        print 'None'
    else:
        print cmod.__name__
```

```
def find_diff():
    from astropy.utils import find_current_module
    print find_current_module(0,True).__name__
```

mod2.py:

```
def find():
    from .mod1 import find2
    find2()
```

With these modules in place, the following occurs:

```
>>> from pkg import mod1, mod2
>>> from astropy.utils import find_current_module
>>> mod1.find1()
pkg.mod1
>>> mod1.find2()
None
>>> mod2.find()
pkg.mod2
>>> find_current_module(0)
<module 'astropy.utils.misc' from 'astropy/utils/misc.py'>
>>> mod1.find_diff()
pkg.mod1
```

isiterable

astropy.utils.misc.isiterable(*obj*)

Returns True if the given object is iterable.

deprecated

astropy.utils.misc.deprecated(*since*, *message=''*, *name=''*, *alternative=''*, *pending=False*, *obj_type='function'*)

Used to mark a function as deprecated.

To mark an attribute as deprecated, use deprecated_attribute.

Parameters

since : str

The release at which this API became deprecated. This is required.

message : str, optional

Override the default deprecation message. The format specifier %(func)s may be used for the name of the function, and %(alternative)s may be used in the deprecation message to insert the name of an alternative to the deprecated function. %(obj_type) may be used to insert a friendly name for the type of object being deprecated.

name : str, optional

The name of the deprecated function; if not provided the name is automatically determined from the passed in function, though this is useful in the case of renamed functions, where the new function is just assigned to the name of the deprecated function. For example:

```
def new_function():
    ...
oldFunction = new_function
```

alternative : str, optional

An alternative function that the user may use in place of the deprecated function. The deprecation warning will tell the user about this alternative if provided.

pending : bool, optional

If True, uses a PendingDeprecationWarning instead of a DeprecationWarning.

deprecated_attribute

`astropy.utils.misc.deprecated_attribute`(*name*, *since*, *message=None*, *alternative=None*, *pending=False*)

Used to mark a public attribute as deprecated. This creates a property that will warn when the given attribute name is accessed. To prevent the warning (i.e. for internal code), use the private name for the attribute by prepending an underscore (i.e. `self._name`).

Parameters

name : str

The name of the deprecated attribute.

since : str

The release at which this API became deprecated. This is required.

message : str, optional

Override the default deprecation message. The format specifier `%(name)s` may be used for the name of the attribute, and `%(alternative)s` may be used in the deprecation message to insert the name of an alternative to the deprecated function.

alternative : str, optional

An alternative attribute that the user may use in place of the deprecated attribute. The deprecation warning will tell the user about this alternative if provided.

pending : bool, optional

If True, uses a PendingDeprecationWarning instead of a DeprecationWarning.

Examples

```
class MyClass:
    # Mark the old_name as deprecated
    old_name = misc.deprecated_attribute('old_name', '0.1')

    def method(self):
        self._old_name = 42
```

format_exception

`astropy.utils.misc.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. For example:

try:
1/0

except:

raise ZeroDivisionError(

format_except('A divide by zero occurred in {filename} at '
'line {lineno} of function {func}.'))

Any additional positional or keyword arguments passed to this function are also used to format the message.

Note: This uses `sys.exc_info` to gather up the information needed to fill in the formatting arguments. Python 2.x and 3.x have slightly different behavior regarding `sys.exc_info` (the latter will not carry it outside a handled exception), so it's not wise to use this outside of an `except` clause - if it is, this will substitute '<unkown>' for the 4 formatting arguments.

Classes

`lazyproperty`	Works similarly to property(), but computes the value only once.
`NumpyRNGContext`(seed)	A context manager (for use with the `with` statement) that will seed the numpy random number generator (

misc.lazyproperty

`misc.lazyproperty`

Works similarly to property(), but computes the value only once.

This essentially memoizes the value of the property by storing the result of its computation in the `__dict__` of the object instance. This is useful for computing the value of some property that should otherwise be invariant. For example:

```
>>> class LazyTest(object):
...     @lazyproperty
...     def complicated_property(self):
...         print 'Computing the value for complicated_property...'
...         return 42
...
>>> lt = LazyTest()
>>> lt.complicated_property
Computing the value for complicated_property...
42
>>> lt.complicated_property
42
```

If a setter for this property is defined, it will still be possible to manually update the value of the property, if that capability is desired.

Adapted from the recipe at http://code.activestate.com/recipes/363602-lazy-property-evaluation

NumpyRNGContext

class `astropy.utils.misc.NumpyRNGContext`(*seed*)

Bases: `object`

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.

This is primarily intended for use in the astropy testing suit, but it may be useful in ensuring reproducibility of Monte Carlo simulations in a science context.

Parameters

seed : int

The value to use to seed the numpy RNG

Examples

A typical use case might be:

```
with NumpyRNGContext(<some seed value you pick>):
    from numpy import random

    randarr = random.randn(100)
    ... run your test using 'randarr' ...

#Any code using numpy.random at this indent level will act just as it
#would have if it had been before the with statement - e.g. whatever
#the default seed is.
```

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.

HomogeneousList

class astropy.utils.collections.HomogeneousList(*types*, *values*=[ ])

Bases: list

A subclass of list that contains only elements of a given type or types. If an item that is not of the specified type is added to the list, a TypeError is raised.

Methods Summary

insert(i, x)	
extend(x)	
append(x)	

Methods Documentation

insert(*i*, *x*)

extend(*x*)

append(*x*)

astropy.utils.console Module

Utilities for console input and output.

Functions

isatty(file)	Returns True if file is a tty.
color_print(*args, **kwargs)	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.
print_code_line(line[, col, file, tabwidth, ...])	Prints a line of source code, highlighting a particular character position in the line.

isatty

astropy.utils.console.isatty(*file*)

Returns True if file is a tty.

Most built-in Python file-like objects have an isatty member, but some user-defined types may not, so this assumes those are not ttys.

color_print

astropy.utils.console.color_print(**args*, ***kwargs*)

Prints colors and styles to the terminal uses ANSI escape sequences.

```
color_print('This is the color ', 'default', 'GREEN', 'green')
```

Parameters

positional args : strings

The positional arguments come in pairs (*msg*, *color*), where *msg* is the string to display and *color* is the color to display it in.

color is an ANSI terminal color name. Must be one of: black, red, green, brown, blue, magenta, cyan, lightgrey, default, darkgrey, lightred, lightgreen, yellow, lightblue, lightmagenta, lightcyan, white, or '' (the empty string).

file : writeable file-like object, optional

Where to write to. Defaults to sys.stdout. If file is not a tty (as determined by calling its isatty member, if one exists), no coloring will be included.

end : str, optional

The ending of the message. Defaults to \n. The end will be printed after resetting any color or font state.

human_time

astropy.utils.console.human_time(*seconds*)

Returns a human-friendly time string that is always exactly 6 characters long.

Depending on the number of seconds given, can be one of:

```
1w 3d
2d 4h
1h 5m
1m 4s
  15s
```

Will be in color if console coloring is turned on.

Parameters

seconds : int

The number of seconds to represent

Returns

time : str

A human-friendly representation of the given number of seconds that is always exactly 6 characters.

print_code_line

astropy.utils.console.print_code_line(*line*, *col=None*, *file=<open file '<stdout>', mode 'w' at 0x7f23551061e0>*, *tabwidth=8*, *width=70*)

Prints a line of source code, highlighting a particular character position in the line. Useful for displaying the context of error messages.

If the line is more than `width` characters, the line is truncated accordingly and '...' characters are inserted at the front and/or end.

It looks like this:

```
there_is_a_syntax_error_here :
                             ^
```

Parameters

line : unicode

The line of code to display

col : int, optional

The character in the line to highlight. `col` must be less than `len(line)`.

file : writeable file-like object, optional

Where to write to. Defaults to `sys.stdout`.

tabwidth : int, optional

The number of spaces per tab (`'\t'`) character. Default is 8. All tabs will be converted to spaces to ensure that the caret lines up with the correct column.

width : int, optional

The width of the display, beyond which the line will be truncated. Defaults to 70 (this matches the default in the standard library's `textwrap` module).

Classes

ProgressBar(total[, file])	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

ProgressBar

class astropy.utils.console.ProgressBar(*total*, *file=<open file '<stdout>', mode 'w' at 0x7f23551061e0>*)

A class to display a progress bar in the terminal.

It is designed to be used with the `with` statement:

```
with ProgressBar(len(items)) as bar:
    for item in enumerate(items):
        bar.update()
```

Methods Summary

map(function, items[, multiprocess, file])	Does a map operation while displaying a progress bar with percentage complete.
update([value])	Update the progress bar to the given value (out of the total given to the constructor.
iterate(items[, file])	Iterate over a sequence while indicating progress with a progress bar in the terminal.

Methods Documentation

classmethod map(*function*, *items*, *multiprocess=False*, *file=<open file '<stdout>', mode 'w' at 0x7f23551061e0>*)

Does a map operation while displaying a progress bar with percentage complete.

```
def work(i):
    print(i)

ProgressBar.map(work, range(50))
```

Parameters

function : function

Function to call for each step

items : sequence

Sequence where each element is a tuple of arguments to pass to *function*.

multiprocess : bool, optional

If `True`, use the `multiprocessing` module to distribute each task to a different processor core.

file : writeable file-like object, optional

The file to write the progress bar to. Defaults to `sys.stdout`. If `file` is not a tty (as determined by calling its `isatty` member, if any), the scrollbar will be completely silent.

update(*value=None*)
Update the progress bar to the given value (out of the total given to the constructor.

classmethod iterate(*items*, *file=<open file '<stdout>', mode 'w' at 0x7f23551061e0>*)
Iterate over a sequence while indicating progress with a progress bar in the terminal.

```
for item in ProgressBar.iterate(items):
    pass
```

Parameters
items : sequence

A sequence of items to iterate over

file : writeable file-like object, optional

The file to write the progress bar to. Defaults to sys.stdout. If file is not a tty (as determined by calling its isatty member, if any), the scrollbar will be completely silent.

Returns
generator : :

A generator over items

Spinner

class astropy.utils.console.Spinner(*msg*, *color='default'*, *file=<open file '<stdout>', mode 'w' at 0x7f23551061e0>*, *step=1*, *chars=None*)
A class to display a spinner in the terminal.

It is designed to be used with the with statement:

```
with Spinner("Reticulating splines", "green") as s:
    for item in enumerate(items):
        s.next()
```

ProgressBarOrSpinner

class astropy.utils.console.ProgressBarOrSpinner(*total*, *msg*, *color='default'*, *file=<open file '<stdout>', mode 'w' at 0x7f23551061e0>*)
A class that displays either a ProgressBar or Spinner depending on whether the total size of the operation is known or not.

It is designed to be used with the with statement:

```
if file.has_length():
    length = file.get_length()
else:
    length = None
bytes_read = 0
with ProgressBarOrSpinner(length) as bar:
    while file.read(blocksize):
        bytes_read += blocksize
        bar.update(bytes_read)
```

Methods Summary

update(value)	Update the progress bar to the given value (out of the total given to the constructor.

Methods Documentation

update(*value*)
Update the progress bar to the given value (out of the total given to the constructor.

File Downloads

astropy.utils.data Module

This module contains helper functions for accessing, downloading, and caching data files.

Functions

get_readable_fileobj(*args, **kwds)	Given a filename or a readable file-like object, return a context manager that
get_file_contents(name_or_obj[, encoding, cache])	Retrieves the contents of a filename or file-like object.
get_pkg_data_fileobj(data_name[, encoding, ...])	Retrieves a data file from the standard locations for the package and provide
get_pkg_data_filename(data_name)	Retrieves a data file from the standard locations for the package and provide
get_pkg_data_contents(data_name[, encoding, ...])	Retrieves a data file from the standard locations and returns its contents as a
get_pkg_data_fileobjs(datadir[, pattern, ...])	Returns readable file objects for all of the data files in a given directory that
get_pkg_data_filenames(datadir[, pattern])	Returns the path of all of the data files in a given directory that match a give
compute_hash(localfn)	Computes the MD5 hash for a file.
clear_download_cache([hashorurl])	Clears the data file cache by deleting the local file(s).
download_file(remote_url[, cache])	Accepts a URL, downloads and optionally caches the result returning the fil

get_readable_fileobj
astropy.utils.data.get_readable_fileobj(**args*, ***kwds*)
Given a filename or a readable file-like object, return a context manager that yields a readable file-like object.

This supports passing filenames, URLs, and readable file-like objects, any of which can be compressed in gzip or bzip2.

Parameters

name_or_obj : str or file-like object

The filename of the file to access (if given as a string), or the file-like object to access.

If a file-like object, it must be opened in binary mode.

encoding : str, optional

When None (default), returns a file-like object with a read method that on Python 2.x returns bytes objects and on Python 3.x returns str (unicode) objects, using locale.getpreferredencoding() as an encoding. This matches the default behavior of the built-in open when no mode argument is provided.

When 'binary', returns a file-like object where its read method returns bytes objects.

When another string, it is the name of an encoding, and the file-like object's read method will return str (unicode) objects, decoded from binary using the given

encoding.

cache : bool, optional

Whether to cache the contents of remote URLs

Notes

This function is a context manager, and should be used for example as:

```
with get_readable_fileobj('file.dat') as f:
    contents = f.read()
```

get_file_contents

`astropy.utils.data.get_file_contents`(*name_or_obj*, *encoding=None*, *cache=False*)

Retrieves the contents of a filename or file-like object.

See the `get_readable_fileobj` docstring for details on parameters.

Returns

content :

The content of the file (as requested by `encoding`).

get_pkg_data_fileobj

`astropy.utils.data.get_pkg_data_fileobj`(*data_name*, *encoding=None*, *cache=True*)

Retrieves a data file from the standard locations for the package and provides the file as a file-like object that reads bytes.

Parameters

data_name : str

Name/location of the desired data file. One of the following:

- The name of a data file included in the source distribution. The path is relative to the module calling this function. For example, if calling from `astropy.pkname`, use `'data/file.dat'` to get the file in `astropy/pkgname/data/file.dat`. Double-dots can be used to go up a level. In the same example, use `'../data/file.dat'` to get `astropy/data/file.dat`.
- If a matching local file does not exist, the Astropy data server will be queried for the file.
- A hash like that produced by `compute_hash` can be requested, prefixed by 'hash/' e.g. 'hash/395dd6493cc584df1e78b474fb150840'. The hash will first be searched for locally, and if not found, the Astropy data server will be queried.

encoding : str, optional

When `None` (default), returns a file-like object with a `read` method that on Python 2.x returns `bytes` objects and on Python 3.x returns `str` (`unicode`) objects, using `locale.getpreferredencoding()` as an encoding. This matches the default behavior of the built-in `open` when no `mode` argument is provided.

When `'binary'`, returns a file-like object where its `read` method returns `bytes` objects.

When another string, it is the name of an encoding, and the file-like object's `read` method will return `str` (`unicode`) objects, decoded from binary using the given encoding.

cache : bool

If True, the file will be downloaded and saved locally or the already-cached local copy will be accessed. If False, the file-like object will directly access the resource (e.g. if a remote URL is accessed, an object like that from `urllib2.urlopen` is returned).

Returns

fileobj : file-like

An object with the contents of the data file available via `read()`. Can be used as part of a `with` statement, automatically closing itself after the `with` block.

Raises

urllib2.URLError :

If a remote file cannot be found.

IOError :

If problems occur writing or reading a local file.

See Also:

`get_pkg_data_contents`
returns the contents of a file or url as a bytes object

`get_pkg_data_filename`
returns a local name for a file containing the data

Examples

This will retrieve a data file and its contents for the `astropy.wcs` tests:

```
from astropy.config import get_pkg_data_fileobj

with get_pkg_data_fileobj('data/3d_cd.hdr') as fobj:
    fcontents = fobj.read()
```

This would download a data file from the astropy data server because the `standards/vega.fits` file is not present in the source distribution. It will also save the file locally so the next time it is accessed it won't need to be downloaded.:

```
from astropy.config import get_pkg_data_fileobj

with get_pkg_data_fileobj('standards/vega.fits') as fobj:
    fcontents = fobj.read()
```

This does the same thing but does *not* cache it locally:

```
with get_pkg_data_fileobj('standards/vega.fits', cache=False) as fobj:
    fcontents = fobj.read()
```

get_pkg_data_filename
astropy.utils.data.get_pkg_data_filename(*data_name*)

Retrieves a data file from the standard locations for the package and provides a local filename for the data.

This function is similar to get_pkg_data_fileobj but returns the file *name* instead of a readable file-like object. This means that this function must always cache remote files locally, unlike get_pkg_data_fileobj.

Parameters

data_name : str

Name/location of the desired data file. One of the following:

- The name of a data file included in the source distribution. The path is relative to the module calling this function. For example, if calling from astropy.pkname, use 'data/file.dat' to get the file in astropy/pkgname/data/file.dat. Double-dots can be used to go up a level. In the same example, use '../data/file.dat' to get astropy/data/file.dat.
- If a matching local file does not exist, the Astropy data server will be queried for the file.
- A hash like that produced by compute_hash can be requested, prefixed by 'hash/' e.g. 'hash/395dd6493cc584df1e78b474fb150840'. The hash will first be searched for locally, and if not found, the Astropy data server will be queried.

Returns

filename : str

A file path on the local file system corresponding to the data requested in data_name.

Raises

urllib2.URLError :

If a remote file cannot be found.

IOError :

If problems occur writing or reading a local file.

See Also:

get_pkg_data_contents
: returns the contents of a file or url as a bytes object

get_pkg_data_fileobj
: returns a file-like object with the data

Examples

This will retrieve the contents of the data file for the astropy.wcs tests:

```
from astropy.config import get_pkg_data_filename

fn = get_pkg_data_filename('data/3d_cd.hdr')
with open(fn) as f:
    fcontents = f.read()
```

This retrieves a data file by hash either locally or from the astropy data server:

```
from astropy.config import get_pkg_data_filename

fn = get_pkg_data_filename('hash/da34a7b07ef153eede67387bf950bb32')
with open(fn) as f:
    fcontents = f.read()
```

get_pkg_data_contents

`astropy.utils.data.get_pkg_data_contents`(*data_name*, *encoding=None*, *cache=True*)

Retrieves a data file from the standard locations and returns its contents as a bytes object.

Parameters

data_name : str

Name/location of the desired data file. One of the following:

- The name of a data file included in the source distribution. The path is relative to the module calling this function. For example, if calling from `astropy.pkname`, use `'data/file.dat'` to get the file in `astropy/pkgname/data/file.dat`. Double-dots can be used to go up a level. In the same example, use `'../data/file.dat'` to get `astropy/data/file.dat`.
- If a matching local file does not exist, the Astropy data server will be queried for the file.
- A hash like that produced by `compute_hash` can be requested, prefixed by 'hash/' e.g. 'hash/395dd6493cc584df1e78b474fb150840'. The hash will first be searched for locally, and if not found, the Astropy data server will be queried.
- A URL to some other file.

encoding : str, optional

When `None` (default), returns a file-like object with a `read` method that on Python 2.x returns `bytes` objects and on Python 3.x returns `str` (`unicode`) objects, using `locale.getpreferredencoding()` as an encoding. This matches the default behavior of the built-in `open` when no `mode` argument is provided.

When `'binary'`, returns a file-like object where its `read` method returns `bytes` objects.

When another string, it is the name of an encoding, and the file-like object's `read` method will return `str` (`unicode`) objects, decoded from binary using the given encoding.

cache : bool

If True, the file will be downloaded and saved locally or the already-cached local copy will be accessed. If False, the file-like object will directly access the resource (e.g. if a remote URL is accessed, an object like that from `urllib2.urlopen` is returned).

Returns

contents : bytes

The complete contents of the file as a bytes object.

Raises

urllib2.URLError :

If a remote file cannot be found.

IOError :

If problems occur writing or reading a local file.

See Also:

`get_pkg_data_fileobj`
returns a file-like object with the data
`get_pkg_data_filename`
returns a local name for a file containing the data

get_pkg_data_fileobjs
`astropy.utils.data.get_pkg_data_fileobjs(`*datadir*, *pattern='*'*, *encoding=None*`)`
Returns readable file objects for all of the data files in a given directory that match a given glob pattern.

Parameters

datadir : str

Name/location of the desired data files. One of the following:

- The name of a directory included in the source distribution. The path is relative to the module calling this function. For example, if calling from `astropy.pkname`, use `'data'` to get the files in `astropy/pkgname/data`
- Remote URLs are not currently supported

pattern : str, optional

A UNIX-style filename glob pattern to match files. See the `glob` module in the standard library for more information. By default, matches all files.

encoding : str, optional

When `None` (default), returns a file-like object with a `read` method that on Python 2.x returns `bytes` objects and on Python 3.x returns `str` (`unicode`) objects, using `locale.getpreferredencoding()` as an encoding. This matches the default behavior of the built-in `open` when no `mode` argument is provided.

When `'binary'`, returns a file-like object where its `read` method returns `bytes` objects.

When another string, it is the name of an encoding, and the file-like object's `read` method will return `str` (`unicode`) objects, decoded from binary using the given encoding.

Returns

fileobjs : iterator of file objects

File objects for each of the files on the local filesystem in *datadir* matching *pattern*.

Examples

This will retrieve the contents of the data file for the `astropy.wcs` tests:

```
from astropy.config import get_pkg_data_filenames

for fd in get_pkg_data_filename('maps', '*.hdr'):
    fcontents = fd.read()
```

get_pkg_data_filenames
astropy.utils.data.get_pkg_data_filenames(*datadir*, *pattern='*'*)
Returns the path of all of the data files in a given directory that match a given glob pattern.

Parameters

datadir : str

Name/location of the desired data files. One of the following:

- The name of a directory included in the source distribution. The path is relative to the module calling this function. For example, if calling from astropy.pkname, use 'data' to get the files in astropy/pkgname/data.
- Remote URLs are not currently supported.

pattern : str, optional

A UNIX-style filename glob pattern to match files. See the glob module in the standard library for more information. By default, matches all files.

Returns

filenames : iterator of str

Paths on the local filesystem in *datadir* matching *pattern*.

Examples

This will retrieve the contents of the data file for the astropy.wcs tests:

```
from astropy.config import get_pkg_data_filenames

for fn in get_pkg_data_filename('maps', '*.hdr'):
    with open(fn) as f:
        fcontents = f.read()
```

compute_hash
astropy.utils.data.compute_hash(*localfn*)
Computes the MD5 hash for a file.

The hash for a data file is used for looking up data files in a unique fashion. This is of particular use for tests; a test may require a particular version of a particular file, in which case it can be accessed via hash to get the appropriate version.

Typically, if you wish to write a test that requires a particular data file, you will want to submit that file to the astropy data servers, and use e.g. get_pkg_data_filename('hash/a725fa6ba642587436612c2df0451956'), but with the hash for your file in place of the hash in the example.

Parameters

localfn : str

The path to the file for which the hash should be generated.

Returns

md5hash : str

The hex digest of the MD5 hash for the contents of the localfn file.

clear_download_cache
astropy.utils.data.clear_download_cache(*hashorurl=None*)
Clears the data file cache by deleting the local file(s).

Parameters

hashorurl : str or None

If None, the whole cache is cleared. Otherwise, either specifies a hash for the cached file that is supposed to be deleted, or a URL that has previously been downloaded to the cache.

Raises

OSEerror :

If the requested filename is not present in the data directory.

download_file
astropy.utils.data.download_file(*remote_url*, *cache=False*)
Accepts a URL, downloads and optionally caches the result returning the filename, with a name determined by the file's MD5 hash. If cache=True and the file is present in the cache, just returns the filename.

Parameters

remote_url : str

The URL of the file to download

cache : bool, optional

Whether to use the cache

Classes

CacheMissingWarning	This warning indicates the standard cache directory is not accessible, with the first argument providing the

CacheMissingWarning
exception astropy.utils.data.CacheMissingWarning
This warning indicates the standard cache directory is not accessible, with the first argument providing the warning message. If args[1] is present, it is a filename indicating the path to a temporary file that was created to store a remote data download in the absence of the cache.

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.

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 de
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.

Functions

check_anyuri
astropy.utils.xml.check.check_anyuri(*uri*)
Returns True if *uri* is a valid URI as defined in RFC 2396.

check_id
astropy.utils.xml.check.check_id(*ID*)
Returns True if *ID* is a valid XML ID.

check_mime_content_type
astropy.utils.xml.check.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
astropy.utils.xml.check.check_token(*token*)
Returns True if *token* is a valid XML token, as defined by XML Schema Part 2.

fix_id
astropy.utils.xml.check.fix_id(*ID*)
Given an arbitrary string, create one that can be used as an xml id. This is rather simplistic at the moment, since it just replaces non-valid characters with underscores.

astropy.utils.xml.iterparser Module

This module includes a fast iterator-based XML parser.

get_xml_iterator(*args, **kwds)	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.

Functions

get_xml_iterator
astropy.utils.xml.iterparser.get_xml_iterator(**args*, ***kwds*)
Returns an iterator over the elements of an XML file.

The iterator doesn't ever build a tree, so it is much more memory and time efficient than the alternative in cElementTree.

Parameters
fd : readable file-like object or read function

Returns
parts : iterator

The iterator returns 4-tuples (*start*, *tag*, *data*, *pos*):

- *start*: when True is a start element event, otherwise an end element event.
- *tag*: The name of the element

- *data*: Depends on the value of *event*:
  - if *start* == `True`, data is a dictionary of attributes
  - if *start* == `False`, data is a string containing the text content of the element
- *pos*: Tuple (*line*, *col*) indicating the source of the event.

get_xml_encoding

`astropy.utils.xml.iterparser.get_xml_encoding(`*source*`)`

Determine the encoding of an XML file by reading its header.

Parameters

source : readable file-like object, read function or str path

Returns

encoding : str

xml_readlines

`astropy.utils.xml.iterparser.xml_readlines(`*source*`)`

Get the lines from a given XML file. Correctly determines the encoding and always returns unicode.

Parameters

source : readable file-like object, read function or str path

Returns

lines : list of unicode

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.

`validate_schema`(filename, schema_file)	Validates an XML file against a schema or DTD.

Functions

validate_schema

`astropy.utils.xml.validate.validate_schema(`*filename*, *schema_file*`)`

Validates an XML file against a schema or DTD.

Parameters

filename : str

The path to the XML file to validate

schema : str

The path to the XML schema or DTD

Returns

returncode, stdout, stderr : int, str, str

Returns the returncode from xmllint and the stdout and stderr as strings

astropy.utils.xml.writer Module

Contains a class that makes it simple to stream out well-formed and nicely-indented XML.

xml_escape(s)	Escapes &, ‘, ”, < and > in an XML attribute value.
xml_escape_cdata(s)	Escapes &, < and > in an XML CDATA string.

Functions

xml_escape
astropy.utils.xml.writer.xml_escape(*s*)
Escapes &, ‘, ”, < and > in an XML attribute value.

xml_escape_cdata
astropy.utils.xml.writer.xml_escape_cdata(*s*)
Escapes &, < and > in an XML CDATA string.

XMLWriter(file)	A class to write well-formed and nicely indented XML.

Classes

XMLWriter
class astropy.utils.xml.writer.XMLWriter(*file*)
A class to write well-formed and nicely indented XML.

Use like this:

```
w = XMLWriter(fh)
with w.tag('html'):
    with w.tag('body'):
        w.data('This is the content')
```

Which produces:

```
<html>
 <body>
  This is the content
 </body>
</html>
```

Methods Summary

comment(comment)	Adds a comment to the output stream.
end([tag, indent, wrap])	Closes the current element (opened by the most recent call to start).
start(tag[, attrib])	Opens a new element.

Table 1.203 – continued from previous page

get_indentation_spaces([offset])	Returns a string of spaces that matches the current indentation level.
element(tag[, text, wrap, attrib])	Adds an entire element.
tag(*args, **kwds)	A convenience method for use with the with statement:: with writer.tag('foo'): writer.element(
object_attrs(obj, attrs)	Converts an object with a bunch of attributes on an object into a dictionary for use by the XMLW
flush()	
close(id)	Closes open elements, up to (and including) the element identified by the given identifier.
get_indentation()	Returns the number of indentation levels the file is currently in.
data(text)	Adds character data to the output stream.

Methods Documentation

comment(*comment*)
Adds a comment to the output stream.

Parameters
comment : str

Comment text, as a Unicode string.

end(*tag=None*, *indent=True*, *wrap=False*)
Closes the current element (opened by the most recent call to start).

Parameters
tag : str

Element name. If given, the tag must match the start tag. If omitted, the current element is closed.

start(*tag*, *attrib={}*, ***extra*)
Opens a new element. Attributes can be given as keyword arguments, or as a string/string dictionary. The method returns an opaque identifier that can be passed to the close() method, to close all open elements up to and including this one.

Parameters
tag : str

The element name

attrib : dict of str -> str

Attribute dictionary. Alternatively, attributes can be given as keyword arguments.

Returns
id : int

Returns an element identifier.

get_indentation_spaces(*offset=0*)
Returns a string of spaces that matches the current indentation level.

element(*tag*, *text=None*, *wrap=False*, *attrib={}*, ***extra*)
Adds an entire element. This is the same as calling start, data, and end in sequence. The text argument can be omitted.

tag(**args*, ***kwds*)
A convenience method for use with the with statement:

```
with writer.tag('foo'):
    writer.element('bar')
# </foo> is implicitly closed here
```

Parameters are the same as to `start`.

static `object_attrs`(*obj*, *attrs*)
Converts an object with a bunch of attributes on an object into a dictionary for use by the `XMLWriter`.

Parameters
obj : object

Any Python object

attrs : sequence of str

Attribute names to pull from the object

Returns
attrs : dict

Maps attribute names to the values retrieved from `obj.attr`. If any of the attributes is `None`, it will not appear in the output dictionary.

`flush`()

`close`(*id*)
Closes open elements, up to (and including) the element identified by the given identifier.

Parameters
id : int

Element identifier, as returned by the `start` method.

`get_indentation`()
Returns the number of indentation levels the file is currently in.

`data`(*text*)
Adds character data to the output stream.

Parameters
text : str

Character data, as a Unicode string.

1.17 Configuration system (`astropy.config`)

1.17.1 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.

1.17.2 Getting Started

To see the configuration options, look for your astropy configuration file. You can find it by doing:

```
from astropy.config import get_config_dir

print get_config_dir()
```

And you should see the location of your configuration directory. The standard scheme generally puts your configuration directory in `$HOME/.astropy/config`, but if you've set the environment variable `XDG_CONFIG_HOME` and the `$XDG_CONFIG_HOME/astropy` directory exists, it will instead be there.

Once you've 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 the 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, just do:

```
from astropy.config import reload_config

reload_config()
```

Warning: The above is not true yet, because the setup doesn't automatically populate the configuration files. Hopefully it will be true soon, though. The `_generate_all_config_items()` function will already do this, basically, but there has to be some thought about how to make driver scripts that actually do this for each user, and coordinate when they get run so that everything is already built.

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.

1.17.3 Using `config`

Changing Values at Run-time

The configuration system is most conveniently used by modifying configuration files as described above. Values can also, however, be modified in an active python session using the `set()` method. A run-time `ConfigurationItem` object can be used to make these changes. These items are found in the same module as the configuration section they are in, and usually have the same name as in the configuration files, but in all caps. Alternatively, they may be located with the `get_config_items()` function.

For 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 run-time this way:

```
>>> from astropy.utils.data import DATAURL, REMOTE_TIMEOUT
>>> DATAURL()
```

```
'http://data.astropy.org/'
>>> DATAURL.set('http://astropydata.mywebsite.com')
>>> DATAURL()
'http://astropydata.mywebsite.com'
>>> REMOTE_TIMEOUT()
3.0
>>> REMOTE_TIMEOUT.set(4.5)
>>> REMOTE_TIMEOUT()
4.5
```

Or alternatively:

```
>>> from astropy.config import get_config

>>> items = get_config('astropy.utils.data')
>>> items['dataurl'].set('http://astropydata.mywebsite.com')
>>> items['remote_timeout'].set('4.5')
```

Note that this will *not* permanently change these values in the configuration files - just for the current session. To change the configuration files, after you've made your changes, you can do:

```
>>> DATAURL.save()
>>> REMOTE_TIMEOUT.save()
```

Or to save all modifications to configuration items in `astropy.utils.data` (which includes the changes made above), do:

```
>>> from astropy.config import save_config
>>> save_config('astropy.utils.data')
```

Reloading Configuration

Instead of modifying the variables in python, you can also modify the configuration files and then reload them. For 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:

```
>>> DATAURL.reload()
>>> REMOTE_TIMEOUT.reload()
```

This should update the variables with the values from the configuration file:

```
>>> DATAURL()
'http://myotherdata.mywebsite.com/'
>>> REMOTE_TIMEOUT()
6.3
```

Or if you want to reload all astropy configuration at once, use the `reload_config` function:

```
>>> config.reload_config('astropy')
```

Developer Usage

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. 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 full interface for a `ConfigurationItem` - this is a guide for *typical* developer usage. In almost all cases, a configuration item should be defined and used in the following manner:

```
""" This is the docstring at the beginning of a module
"""
from astropy.config import ConfigurationItem

SOME_OPTION = ConfigurationItem('some_option', 1, 'A description.')
ANOTHER_OPTION = ConfigurationItem('annother_opt', 'a string val',
                                   'A longer description of what this does.')

... implementation ...
def some_func():
    #to get the value of these options, I might do:
    something = SOME_OPTION() + 2
    return ANOTHER_OPTION() + ' Also, I added text.'
```

It is highly recommended that any configuration items be placed at the top of a module like this, as they can then be easily found when viewing the source code and the automated tools to generate the default configuration files can also locate these items.

There are a couple important gotchas to remember about using configuration items in your code. First, it is tempting to do something like:

```
SOME_OPTION = ConfigurationItem('some_option',1,'A description.')

def some_func():
    return SOME_OPTION + 2  # WRONG, you wanted SOME_OPTION() + 2
```

but this is incorrect, because `SOME_OPTION` instead of `SOME_OPTION()` will yield a `ConfigurationItem` object, instead of the *value* of that item (an integer, in this case).

The second point to keep in mind is that `ConfigurationItem` objects can be changed at runtime by users. So you always read their values instead of just 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:

```
SOME_OPTION = ConfigurationItem('some_option',1,'A description.')

def some_func(val=SOME_OPTION()):
    return val + 2
```

This works fine as long as the user doesn't change its value during runtime, but if they do, the function won't know about the change:

```
>>> some_func()
3
>>> SOME_OPTION.set(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_OPTION' configuration item influences this output
    """
    return SOME_OPTION() + 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_OPTION' configuration item.
    """
    return (SOME_OPTION() if val is None else val) + 2
```

1.17.4 See Also

Logging system (overview of `astropy.logger`)

1.17.5 Reference/API

astropy.config Module

This module contains configuration and setup utilities for the `astropy` project. This includes all functionality related to the affiliated package index.

Functions

`get_cache_dir`()	Determines the Astropy cache directory name and creates the directory if it doesn't exist.
`get_config`([packageormod, reload])	Gets the configuration object or section associated with a particular package or module.
`get_config_dir`([create])	Determines the Astropy configuration directory name and creates the directory if it doesn't ex
`reload_config`([packageormod])	Reloads configuration settings from a configuration file for the root package of the requested
`save_config`([packageormod])	Saves all configuration settings to the configuration file for the root package of the requested

get_cache_dir
`astropy.config.paths.get_cache_dir()`

Determines the Astropy cache directory name and creates the directory if it doesn't exist.

This directory is typically `$HOME/.astropy/cache`, but if the XDG_CACHE_HOME environment variable is set and the `$XDG_CACHE_HOME/astropy` directory exists, it will be that directory. If neither exists, the former will be created and symlinked to the latter.

Returns

cachedir : str

The absolute path to the cache directory.

get_config

`astropy.config.configuration.get_config(`*packageormod=None*, *reload=False*`)`

Gets the configuration object or section associated with a particular package or module.

Parameters

packageormod : str or None

The package for which to retrieve the configuration object. If a string, it must be a valid package name, or if None, the package from which this function is called will be used.

Returns

cfgobj : `configobj.ConfigObj` or `configobj.Section`

If the requested package is a base package, this will be the `configobj.ConfigObj` for that package, or if it is a subpackage or module, it will return the relevant `configobj.Section` object.

Raises

RuntimeError :

If `package` is None, but the package this item is created from cannot be determined.

get_config_dir

`astropy.config.paths.get_config_dir(`*create=True*`)`

Determines the Astropy configuration directory name and creates the directory if it doesn't exist.

This directory is typically `$HOME/.astropy/config`, but if the XDG_CONFIG_HOME environment variable is set and the `$XDG_CONFIG_HOME/astropy` directory exists, it will be that directory. If neither exists, the former will be created and symlinked to the latter.

Returns

configdir : str

The absolute path to the configuration directory.

reload_config

`astropy.config.configuration.reload_config(`*packageormod=None*`)`

Reloads configuration settings from a configuration file for the root package of the requested package/module.

This overwrites any changes that may have been made in `ConfigurationItem` objects. This applies for any items that are based on this file, which is determined by the *root* package of `packageormod` (e.g. 'astropy.cfg' for the 'astropy.config.configuration' module).

Parameters

packageormod : str or None

The package or module name - see `get_config` for details.

save_config

`astropy.config.configuration.save_config(`*packageormod=None*`)`

Saves all configuration settings to the configuration file for the root package of the requested package/module.

This overwrites any configuration items that have been changed in `ConfigurationItem` objects that are based on the configuration file determined by the *root* package of `packageormod` (e.g. 'astropy.cfg' for the 'astropy.config.configuration' module).

Note: To save only a single item, use the `ConfigurationItem.save` method - this will save all options in the current session that may have been changed.

Parameters

packageormod : str or None

The package or module name - see `get_config` for details.

Classes

`ConfigurationItem`(name[, defaultvalue, ...])	A setting and associated value stored in the astropy configuration files.
`ConfigurationMissingWarning`	A Warning that is issued when the configuration directory cannot be accessed (usual
`InvalidConfigurationItemWarning`	A Warning that is issued when the configuration value specified in the astropy config

ConfigurationItem

class `astropy.config.configuration.ConfigurationItem`(*name*, *defaultvalue=''*, *description=None*, *cfgtype=None*, *module=None*)

Bases: `object`

A setting and associated value stored in the astropy configuration files.

These objects are typically defined at the top of astropy subpackages or affiliated packages, and store values or option settings that can be modified by the user to

Parameters

name : str

The (case-sensitive) name of this parameter, as shown in the configuration file.

defaultvalue :

The default value for this item. If this is a list of strings, this item will be interpreted as an 'options' value - this item must be one of those values, and the first in the list will be taken as the default value.

description : str or None

A description of this item (will be shown as a comment in the configuration file)

cfgtype : str or None

A type specifier like those used as the *values* of a particular key in a `configspec` file of `configobj`. If None, the type will be inferred from the default value.

module : str or None

The full module name that this item is associated with. The first element (e.g. 'astropy' if this is 'astropy.config.configuration') will be used to determine the name of the configuration file, while the remaining items determine the section. If None, the package will be inferred from the package within whiich this object's initializer is called.

Raises

RuntimeError :

If `module` is None, but the module this item is created from cannot be determined.

Examples

The following example will create an item 'cfgoption = 42' in the '[configuration]' section of astropy.cfg (located in the directory that `astropy.config.paths.get_config_dir` returns), or if the option is already set, it will take the value from the configuration file:

```
from astropy.config import ConfigurationItem

CFG_OPTION = ConfigurationItem('cfgoption',42,module='astropy.configuration')
```

If called as `CFG_OPTION()`, this will return the value `42`, or some other integer if the `astropy.cfg` file specifies a different value.

If this were a file `astropy/configuration/__init__.py`, the `module` option would not be necessary, as it would automatically detect the correct module.

Methods Summary

`set`(value)	Sets the current value of this `ConfigurationItem`.
`reload`()	Reloads the value of this `ConfigurationItem` from the relevant configuration file.
`save`([value])	Writes a value for this `ConfigurationItem` to the relevant configuration file.
`set_temp`(*args, **kwds)	Sets this item to a specified value only inside a while loop.

Methods Documentation

`set(`*value*`)`

Sets the current value of this `ConfigurationItem`.

This also updates the comments that give the description and type information.

Note: This does *not* save the value of this `ConfigurationItem` to the configuration file. To do that, use `ConfigurationItem.save` or `save_config`.

Parameters

value :

The value this item should be set to.

Raises

TypeError :

If the provided `value` is not valid for this `ConfigurationItem`.

`reload()`

Reloads the value of this `ConfigurationItem` from the relevant configuration file.

Returns

val :

The new value loaded from the configuration file.

`save(`*value=None*`)`

Writes a value for this `ConfigurationItem` to the relevant configuration file.

This also writes updated versions of the comments that give the description and type information.

Note: This only saves the value of this *particular* `ConfigurationItem`. To save all configuration settings for this package at once, see `save_config`.

Parameters
value :

Save this value to the configuration file. If None, the current value of this `ConfigurationItem` will be saved.

Raises
TypeError :

If the provided `value` is not valid for this `ConfigurationItem`.

`set_temp(*args, **kwds)`
Sets this item to a specified value only inside a while loop.

Use as::
ITEM = ConfigurationItem('ITEM', 'default', 'description')

with ITEM.set_temp('newval'):

... do something that wants ITEM's value to be 'newval' ...

ITEM is now 'default' after the with block

Parameters
value :

The value to set this item to inside the with block.

ConfigurationMissingWarning
exception `astropy.config.configuration.ConfigurationMissingWarning`
A Warning that is issued when the configuration directory cannot be accessed (usually due to a permissions problem). If this warning appears, configuration items will be set to their defaults rather than read from the configuration file, and no configuration will persist across sessions.

InvalidConfigurationItemWarning
exception `astropy.config.configuration.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.

1.18 Logging system

1.18.1 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).

1.18.2 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.setColor(False)
```

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.set_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.

1.18.3 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.

1.18.4 Using the configuration file

Options for the logger can be set in the `[config.logging_helper]` section of the Astropy configuration file:

```
[config.logging_helper]

# 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
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'
```

1.18.5 Reference/API

astropy.logger Module

This module defines a logging class based on the built-in logging module

Classes

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 dea

AstropyLogger

class astropy.logger.AstropyLogger(*name*, *level=0*)

Bases: logging.Logger

This class is used to set up the Astropy logging.

The main functionality added by this class over the built-in logging.Logger class is the ability to keep track of the origin of the messages, the ability to enable logging of warnings.warn calls and exceptions, and the addition of colorized output and context managers to easily capture messages to a file or list.

Methods Summary

warnings_logging_enabled()	
exception_logging_enabled()	Determine if the exception-logging mechanism is enabled.
log_to_file(*args, **kwds)	Context manager to temporarily log messages to a file.
disable_color()	Disable colorized output
setLevel(level)	Set the logging level of this logger.
enable_warnings_logging()	Enable logging of warnings.warn() calls Once called, any subsequent calls to warni
enable_color()	Enable colorized output
enable_exception_logging()	Enable logging of exceptions Once called, any uncaught exceptions will be emitted
disable_warnings_logging()	Disable logging of warnings.warn() calls Once called, any subsequent calls to warni
disable_exception_logging()	Disable logging of exceptions Once called, any uncaught exceptions will no longer b
log_to_list(*args, **kwds)	Context manager to temporarily log messages to a list.
makeRecord(name, level, pathname, lineno, ...)	

Methods Documentation

warnings_logging_enabled()

exception_logging_enabled()

Determine if the exception-logging mechanism is enabled.

Returns

exclog : bool

True if exception logging is on, False if not.

log_to_file(**args*, ***kwds*)

Context manager to temporarily log messages to a file.

Parameters

filename : str

The file to log messages to.

filter_level : str

If set, any log messages less important than `filter_level` will not be output to the file. Note that this is in addition to the top-level filtering for the logger, so if the logger has level 'INFO', then setting `filter_level` to `INFO` or `DEBUG` will have no effect, since these messages are already filtered out.

filter_origin : str

If set, only log messages with an origin starting with `filter_origin` will be output to the file.

Notes

By default, the logger already outputs log messages to a file set in the Astropy configuration file. Using this context manager does not stop log messages from being output to that file, nor does it stop log messages from being printed to standard output.

Examples

The context manager is used as:

```
with logger.log_to_file('myfile.log'):
    # your code here
```

`disable_color()`
Disable colorized output

`setLevel(`*level*`)`
Set the logging level of this logger.

`enable_warnings_logging()`
Enable logging of warnings.warn() calls

Once called, any subsequent calls to `warnings.warn()` are redirected to this logger and emitted with level `WARN`. Note that this replaces the output from `warnings.warn`.

This can be disabled with `disable_warnings_logging`.

`enable_color()`
Enable colorized output

`enable_exception_logging()`
Enable logging of exceptions

Once called, any uncaught exceptions will be emitted with level `ERROR` by this logger, before being raised.

This can be disabled with `disable_exception_logging`.

`disable_warnings_logging()`
Disable logging of warnings.warn() calls

Once called, any subsequent calls to `warnings.warn()` are no longer redirected to this logger.

This can be re-enabled with `enable_warnings_logging`.

`disable_exception_logging()`
Disable logging of exceptions

Once called, any uncaught exceptions will no longer be emitted by this logger.

This can be re-enabled with `enable_exception_logging`.

`log_to_list(*args, **kwds)`
Context manager to temporarily log messages to a list.

Parameters

filename : str

The file to log messages to.

filter_level : str

If set, any log messages less important than `filter_level` will not be output to the file. Note that this is in addition to the top-level filtering for the logger, so if the logger has level 'INFO', then setting `filter_level` to `INFO` or `DEBUG` will have no effect, since these messages are already filtered out.

filter_origin : str

If set, only log messages with an origin starting with `filter_origin` will be output to the file.

Notes

Using this context manager does not stop log messages from being output to standard output.

Examples

The context manager is used as:

```
with logger.log_to_list() as log_list:
    # your code here
```

`makeRecord(name, level, pathname, lineno, msg, args, exc_info, func=None, extra=None, sinfo=None)`

LoggingError
exception `astropy.logger.LoggingError`
This exception is for various errors that occur in the astropy logger, typically when activating or deactivating logger-related features.

1.19 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 accross sub-packages. This document summarises 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.

Note that until version 1.0, even sub-packages considered *Mature* could undergo some user interface changes as we work to integrate the packages better. Thus, we cannot guarantee complete backward-compatibility between versions at this stage.

The classification is as follows:

The current planned and existing sub-packages are:

1.20 Major Release History

1.20.1 What's New in Astropy 0.2

1.20.2 What's New in Astropy 0.1

This was the initial version of Astropy, released on June 19, 2012. It was released primarily as a "developer preview" for developers interested in working directly on Astropy, on affiliated packages, or on other software that might integrate with Astropy.

Astropy 0.1 integrated several existing packages under a single `astropy` package with a unified installer, including:

- asciitable as `astropy.io.ascii`
- PyFITS as `astropy.io.fits`
- votable as `astropy.io.vo`
- PyWCS as `astropy.wcs`

It also added the beginnings of the `astropy.cosmology` package, and new common data structures for science data in the `astropy.nddata` and `astropy.table` packages.

It also laid much of the groundwork for Astropy's installation and documentation frameworks, as well as tools for managing configuration and data management. These facilities are designed to be shared by Astropy's affiliated packages in the hopes of providing a framework on which other Astronomy-related Python packages can build.

1.21 License and Credits

1.21.1 The Team

The following people have contributed code used as part of the Astropy core package as of the most recent release:

- Tom Aldcroft
- Kyle Barbary
- Paul Barrett
- Erik Bray
- Neil Crighton
- Alex Conley
- Matt Davis
- Christoph Deil
- Michael Droettboom
- Henry Ferguson
- Adam Ginsburg
- Perry Greenfield

- Frédéric Grollier
- Chris Hanley
- JC Hsu
- Wolfgang Kerzendorf
- Roban Kramer
- Prasanth Nair
- Thomas Robitaille
- Adrian Price-Whelan
- Leo Singer
- James Taylor
- Erik Tollerud
- James Turner
- Julien Woillez

1.21.2 License

Part II

Getting help

If you want to get help or discuss issues with other Astropy users, you can sign up for the astropy mailing list. Alternatively, the astropy-dev list is where you should go to discuss more technical aspects of Astropy with the developers.

Part III

Reporting issues

If you have come across something that you believe is a bug, please open a ticket in the Astropy issue tracker, and we will look into it promptly.

Please try to include an example that demonstrates the issue and will allow the developers to reproduce and fix the problem. If you are seeing a crash then frequently it will help to include the full Python stack trace as well as information about your operating system (e.g. MacOSX version or Linux version).

Part IV

Developer Documentation

The developer documentation contains instructions for how to contribute to Astropy or affiliated packages, as well as coding, documentation, and testing guidelines.

CHAPTER

TWO

VISION FOR A COMMON ASTRONOMY PYTHON PACKAGE

The following document summarizes a vision for a common Astronomy Python package, and how we can best all work together to achieve this. In the following document, this common package will be referred to as the core package. This vision is not set in stone, and we are committed to adapting it to whatever process and guidelines work in practice.

The ultimate goal that we seek is a package that would contain much of the core functionality and some common tools required across Astronomy, but not *everything* Astronomers will ever need. The aim is primarily to avoid duplication for common core tasks, and to provide a robust framework upon which to build more complex tools.

Such a common package should not preclude any other Astronomy package from existing, because there will always be more complex and/or specialized tools required. These tools will be able to rely on a single core library for many tasks, and thus reduce the number of dependencies, reduce duplication of functionality, and increase consistency of their interfaces.

2.1 Procedure

With the help of the community, the coordination committee will start by identifying a few of key areas where initial development/consolidation will be needed (such as FITS, WCS, coordinates, tables, photometry, spectra, etc.) and will encourage teams to be formed to build standalone packages implementing this functionality. These packages will be referred to as affiliated packages (meaning that they are intended for future integration in the core package).

A set of requirements will be set out concerning the interfaces and classes/methods that affiliated packages will need to make available in order to ensure consistency between the different components. As the core package grows, new potential areas/components for the core package will be identified. Competition cannot be avoided, and will not be actively discouraged, but whenever possible, developers should strive to work as a team to provide a single and robust affiliated package, for the benefit of the community.

The affiliated packages will be developed outside the core package in independent repositories, which will allow the teams the choice of tool and organization. Once an affiliated package has implemented the desired functionality, and satisfies quality criteria for coding style, documentation, and testing, it will be considered for inclusion in the core package, and further development will be done directly in the core package either via direct access to the repository, or via patches/pull requests (exactly how this will be done will be decided later).

To ensure uniformity across affiliated packages, and to facilitate integration with the core package, developers who wish to submit their affiliated packages for inclusion in the core will need to follow the layout of a 'template' package that will be provided before development starts.

2.2 Dependencies

Affiliated packages should be able to be imported with only the following dependencies:

- The Python Standard Library NumPy, SciPy, and Matplotlib Components already * in the core Astronomy package

Other packages may be used, but must be imported as needed rather than during the initial import of the package.

If a dependency is needed, but is an affiliated package, the dependent package will need to wait until the dependency is integrated into the core package before being itself considered for inclusion. In the mean time, it can make use of the other affiliated package in its current form, or other packages, so as not to stall development. Thus, the first packages to be included in the core will be those only requiring the standard library, NumPy, SciPy, and Matplotlib.

If the required dependency will never be part of a main package, then by default the dependency can be included but should be imported as needed (meaning that it only prevents the importing of that component, not the entire core package), unless a strong case is made and a general consensus is reached by the community that this dependency is important enough to be required at a higher level.

This system means that packages will be integrated into the core package in an order depending on the dependency tree, and also ensures that the interfaces of packages being integrated into the core package are consistent with those already in the core package.

Initially, no dependency on GUI toolkits will be allowed in the core package. If the community reaches agrees on a single toolkit that could be used, then this toolkit will be allowed (but will only be imported as needed).

2.3 Keeping track of affiliated packages

Affiliated packages will be listed in a central location (in addition to PyPI) that will allow an easy installation of all the affiliated packages, for example with a script that will seamlessly download and install all the affiliated packages. The core package will also include mechanisms to facilitate this installation process.

2.4 Existing Packages

Developers who already have existing packages will be encouraged to continue supporting them for the benefit of users until the core library is considered stable, contains this functionality, and is released to the community. Thereafter, developers should encourage users to transition to using the functionality in the core package, and eventually phase out their own packages, unless they provide added value over the core package.

CHAPTER

THREE

CONTRIBUTING TO/DEVELOPING ASTROPY OR AFFILIATED PACKAGES

3.1 Summary

Any contributions to the core Astropy package, whether bug fixes, improvements to the documentation, or new functionality, can be done via *pull requests* on GitHub. The workflow for this is described below.

However, substantial contributions, such as whole new sub-packages, can first be developed as *affiliated packages* then submitted to the Astropy core via pull requests once ready (as described in *Vision for a Common Astronomy Python Package*). Teams working on affiliated packages are free to choose whatever version control system they wish, but ultimately the affiliated package should be merged into a fork of the Astropy repository in order to be submitted as a pull request (this merging can be done either by the team or by one of the core maintainers).

3.2 Getting started with git

The only absolutely necessary configuration step is identifying yourself and your contact info:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

The following sections cover the installation of the git software, the basic configuration, and links to resources to learn more about using git. However, you can also directly go to the GitHub help pages which offer a great introduction to git and GitHub.

3.2.1 Installing git

The instructions here are adapted from http://book.git-scm.com/2_installing_git.html

Debian/Ubuntu

```
sudo apt-get install git-core
```

Fedora

```
sudo yum install git-core
```

MacOS X

There are several ways to install git on Mac. The easiest is to simply download the OS X installer (git-osx-installer). If you have MacPorts installed, you can also do:

```
sudo port install git-core
```

If you have Fink installed, you can do:

```
sudo apt-get install git
```

In addition, you may want to use a GUI to manage your git repositories. A good example of a free Mac GUI is GitX. Other (non-free) examples include Tower and SourceTree. GitHub have also recently released GitHub for Mac.

Windows

Download and install msysGit

3.2.2 Configuring git

Bare Minimum

The only absolutely necessary configuration step is identifying yourself and your contact info:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

Once you've done this, you can actually ignore the rest of the document unless you want to customize the behavior of git.

Overview

Your personal git configurations are saved in the `.gitconfig` file in your home directory.

Here is an example `.gitconfig` file:

```
[user]
        name = Your Name
        email = you@yourdomain.example.com

[alias]
        ci = commit -a
        co = checkout
        st = status
        stat = status
        br = branch
        wdiff = diff --color-words
```

```
[core]
        editor = vim

[merge]
        log = true
```

You can edit this file directly or you can use the `git config --global` command:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
git config --global core.editor vim
git config --global merge.summary true
```

To set up on another computer, you can copy your `~/.gitconfig` file, or run the commands above.

In detail

user.name and user.email

It is good practice to tell git who you are, for labeling any changes you make to the code. The simplest way to do this is from the command line:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

This will write the settings into your git configuration file, which should now contain a user section with your name and email:

```
[user]
      name = Your Name
      email = you@yourdomain.example.com
```

Of course you'll need to replace `Your Name` and `you@yourdomain.example.com` with your actual name and email address.

Aliases

You might well benefit from some aliases to common commands.

For example, you might well want to be able to shorten `git checkout` to `git co`. Or you may want to alias `git diff --color-words` (which gives a nicely formatted output of the diff) to `git wdiff`

The following `git config --global` commands:

```
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
```

```
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
```

will create an `alias` section in your `.gitconfig` file with contents like this:

```
[alias]
        ci = commit -a
        co = checkout
        st = status -a
        stat = status -a
        br = branch
        wdiff = diff --color-words
```

Editor

You may also want to make sure that your editor of choice is used

```
git config --global core.editor vim
```

Merging

To enforce summaries when doing merges (`~/.gitconfig` file again):

```
[merge]
   log = true
```

Or from the command line:

```
git config --global merge.log true
```

Fancy log output

This is a very nice alias to get a fancy log output; it should go in the `alias` section of your `.gitconfig` file:

```
lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)[%an]%Creset' --abbrev
```

You use the alias with:

```
git lg
```

and it gives graph / text output something like this (but with color!):

```
* 6d8e1ee - (HEAD, origin/my-fancy-feature, my-fancy-feature) NF - a fancy file (45 minutes ago) [Matthew Brett]
*   d304a73 - (origin/placeholder, placeholder) Merge pull request #48 from hhuuggoo/master (2 weeks ago) [Jonathan Terho
|\
| * 4aff2a8 - fixed bug 35, and added a test in test_bugfixes (2 weeks ago) [Hugo]
|/
* a7ff2e5 - Added notes on discussion/proposal made during Data Array Summit. (2 weeks ago) [Corran Webster]
* 68f6752 - Initial implimentation of AxisIndexer - uses 'index_by' which needs to be changed to a call on an Axes object
*   376adbd - Merge pull request #46 from terhorst/master (2 weeks ago) [Jonathan Terhorst]
|\
```

```
| * b605216 - updated joshu example to current api (3 weeks ago) [Jonathan Terhorst]
| * 2e991e8 - add testing for outer ufunc (3 weeks ago) [Jonathan Terhorst]
| * 7beda5a - prevent axis from throwing an exception if testing equality with non-axis object (3 weeks ago) [Jonathan Te
| * 65af65e - convert unit testing code to assertions (3 weeks ago) [Jonathan Terhorst]
| *   956fbab - Merge remote-tracking branch 'upstream/master' (3 weeks ago) [Jonathan Terhorst]
| |\
| |/
```

Thanks to Yury V. Zaytsev for posting it.

3.2.3 Git resources

Tutorials and summaries

- GitHub Help has an excellent series of how-to guides.
- learn.github has an excellent series of tutorials
- The pro git book is a good in-depth book on git.
- A git cheat sheet is a page giving summaries of common commands.
- The git user manual
- The git tutorial
- The git community book
- git ready — a nice series of tutorials
- git casts — video snippets giving git how-tos.
- git magic — extended introduction with intermediate detail
- The git parable is an easy read explaining the concepts behind git.
- git foundation expands on the git parable.
- Fernando Perez' git page — Fernando's git page — many links and tips
- A good but technical page on git concepts
- git svn crash course: git for those of us used to subversion

Advanced git workflow

There are many ways of working with git; here are some posts on the rules of thumb that other projects have come up with:

- Linus Torvalds on git management
- Linus Torvalds on linux git workflow . Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

Manual pages online

You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- git add

- git branch
- git checkout
- git clone
- git commit
- git config
- git diff
- git log
- git pull
- git push
- git remote
- git status

3.3 Workflow

The following two sections describe the workflow for the Astropy core package, but teams working on affiliated packages that have chosen to use git are encouraged to also follow these guidelines internally.

3.3.1 Workflow for Developers

In the present document, we refer to the Astropy `master` branch, as the *trunk*.

Creating a fork

You need to do this only once for each package you want to contribute to. The instructions here are very similar to the instructions at http://help.github.com/fork-a-repo/ — please see that page for more details. We're repeating some of it here just to give the specifics for the Astropy project, and to suggest some default names.

Set up and configure a GitHub account

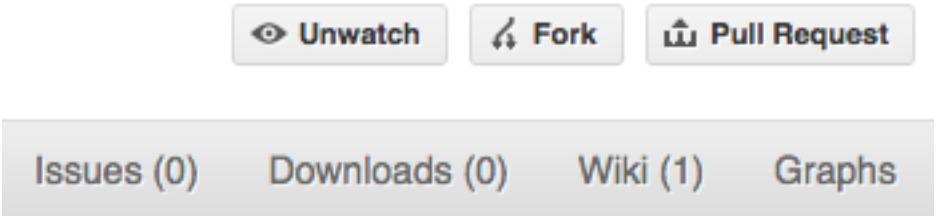
If you don't have a GitHub account, go to the GitHub page, and make one.

You then need to configure your account to allow write access — see the *Generating SSH keys* help on GitHub Help.

Create your own fork of a repository

The following example shows how to fork the core `astropy` repository, but the same applies to other packages:

1. Log into your GitHub account.
2. Go to the Astropy GitHub home.
3. Click on the *fork* button:



Now, after a short pause and some 'Hardcore forking action', you should find yourself at the home page for your own forked copy of Astropy.

Setting up the fork to work on

Overview This is done using:

```
git clone git@github.com:your-user-name/astropy.git
cd astropy
git remote add upstream git://github.com/astropy/astropy.git
```

In detail

1. Clone your fork to the local computer:

   ```
   git clone git@github.com:your-user-name/astropy.git
   ```

2. Change directory to your new repo:

   ```
   cd astropy
   ```

   Then type:

   ```
   git branch -a
   ```

   to show you all branches. You'll get something like:

   ```
   * master
   remotes/origin/master
   ```

   This tells you that you are currently on the `master` branch, and that you also have a `remote` connection to `origin/master`. What remote repository is `remote/origin`? Try `git remote -v` to see the URLs for the remote. They will point to your GitHub fork.

   Now you want to connect to the Astropy repository, so you can merge in changes from the trunk:

   ```
   cd astropy
   git remote add upstream git://github.com/astropy/astropy.git
   ```

   `upstream` here is just the arbitrary name we're using to refer to the main Astropy repository.

   Note that we've used `git://` for the URL rather than `git@`. The `git://` URL is read only. This means we that we can't accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.

Just for your own satisfaction, show yourself that you now have a new 'remote', with `git remote -v show`, giving you something like:

```
upstream   git://github.com/astropy/astropy.git (fetch)
upstream   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)
```

Your fork is now set up correctly, and you are ready to hack away.

Workflow summary

This section gives a summary of the workflow once you have successfully forked the repository, and details are given for each of these steps in the following sections.

- Don't use your `master` branch for anything. Consider deleting it.
- When you are starting a new set of changes, fetch any changes from the trunk, and start a new *feature branch* from that.
- Make a new branch for each separable set of changes — "one task, one branch" (ipython git workflow).
- Name your branch for the purpose of the changes - e.g. `bugfix-for-issue-14` or `refactor-database-code`.
- If you can possibly avoid it, avoid merging trunk or any other branches into your feature branch while you are working.
- If you do find yourself merging from the trunk, consider *Rebasing on trunk*
- Ask on the astropy-dev mailing list if you get stuck.
- Ask for code review!

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.

See linux git workflow and ipython git workflow for some explanation.

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.

Updating the mirror of trunk

From time to time you should fetch the upstream (trunk) changes from GitHub:

```
git fetch upstream
```

This will pull down any commits you don't have, and set the remote branches to point to the right commit. For example, 'trunk' is the branch referred to by (remote/branchname) `upstream/master` - and if there have been commits since you last checked, `upstream/master` will change after you do the fetch.

Making a new feature 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 an 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. For example `add-ability-to-fly`, or `buxfix-for-issue-42`.

```
# Update the mirror of trunk
git fetch upstream

# Make new feature branch starting at current trunk
git branch my-new-feature upstream/master
git checkout my-new-feature
```

Generally, you will want to keep your feature branches on your public GitHub fork. To do this, you git push this new branch up to your github repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your GitHub repo, called `origin`. You push up to your own repo on GitHub with:

```
git push origin my-new-feature
```

In git >= 1.7 you can ensure that the link is correctly set by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

From now on git will know that `my-new-feature` is related to the `my-new-feature` branch in the GitHub repo.

The editing workflow

Overview

```
git add my_new_file
git commit -am 'NF - some message'
git push
```

In more detail

1. Make some changes
2. See which files have changed with `git status` (see git status). You'll see a listing like this one:

```
# On branch ny-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#    modified:   README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
```

```
#    INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```

3. Check what the actual changes are with `git diff` (git diff).
4. Add any new files to version control `git add new_file_name` (see git add).
5. Add any modified files that you want to commit using `git add modified_file_name` (see git add).
6. Once you are ready to commit, check with `git status` which files are about to be committed:

```
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    modified:   README
```

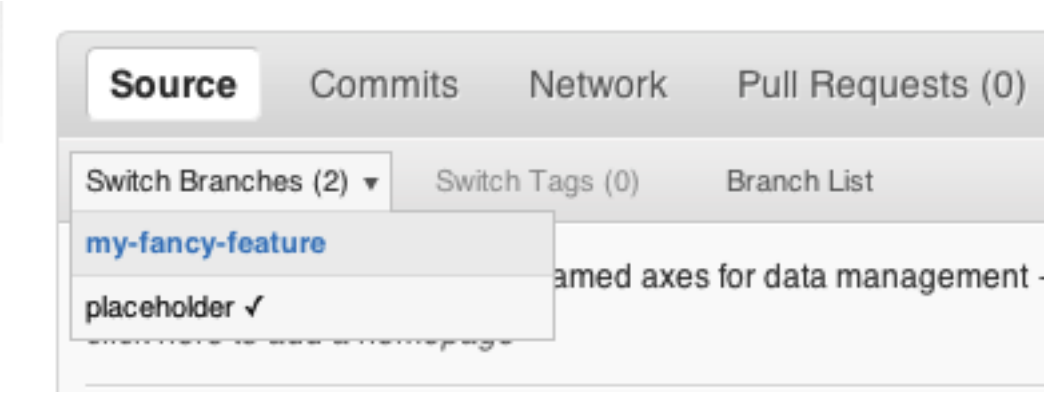
   Then use `git commit -m 'A commit message'`. The `m` flag just signals that you're going to type a message on the command line. The git commit manual page might also be useful.

7. To push the changes up to your forked repo on GitHub, do a `git push` (see git push).

Asking for your changes to be reviewed or merged

When you are ready to ask for someone to review your code and consider a merge:

1. Go to the URL of your forked repo, say `http://github.com/your-user-name/astropy`.
2. Use the 'Switch Branches' dropdown menu near the top left of the page 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. Say if there is anything you'd like particular attention for - like a complicated change or some code you are not happy with.

   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 of getting some preliminary code review.

Some other things you might want to do

Delete a branch on GitHub

```
# change to the master branch (if you still have one, otherwise change to
# another branch)
git checkout master

# delete branch locally
git branch -D my-unwanted-branch

# delete branch on GitHub
git push origin :my-unwanted-branch
```

(Note the colon : before `test-branch`. See also: http://github.com/guides/remove-a-remote-branch

Several people sharing a single repository

If you want to work on some stuff with other people, where you are all committing into the same repository, or even the same branch, then just share it via GitHub.

First fork Astropy into your account, as from *Creating a fork*.

Then, go to your forked repository GitHub page, say `http://github.com/your-user-name/astropy`

Click on the 'Admin' button, and add anyone else to the repo as a collaborator:

Now all those people can do:

```
git clone git@githhub.com:your-user-name/astropy.git
```

Remember that links starting with `git@` use the ssh protocol and are read-write; links starting with `git://` are read-only.

Your collaborators can then commit directly into that repo with the usual:

```
git commit -am 'ENH - much better code'
git push origin master # pushes directly into your repo
```

Explore your repository

To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

You can also look at the network graph visualizer for your GitHub repo.

Finally the *Fancy log output* `lg` alias will give you a reasonable text-based graph of the repository.

Rebasing on trunk

Let's say you thought of some work you'd like to do. You *Updating the mirror of trunk* and *Making a new feature branch* called `cool-feature`. At this stage trunk is at some commit, let's call it E. Now you make some new commits on your `cool-feature` branch, let's call them A, B, C. Maybe your changes take a while, or you come back to them after a while. In the meantime, trunk has progressed from commit E to commit (say) G:

```
      A---B---C cool-feature
     /
D---E---F---G trunk
```

At this stage you consider merging trunk into your feature branch, and you remember that this here page sternly advises you not to do that, because the history will get messy. Most of the time you can just ask for a review, and not worry that trunk has got a little ahead. But sometimes, the changes in trunk might affect your changes, and you need to harmonize them. In this situation you may prefer to do a rebase.

Rebase takes your changes (A, B, C) and replays them as if they had been made to the current state of `trunk`. In other words, in this case, it takes the changes represented by A, B, C and replays them on top of G. After the rebase, your history will look like this:

```
             A'--B'--C' cool-feature
            /
D---E---F---G trunk
```

See rebase without tears for more detail.

To do a rebase on trunk:

```
# Update the mirror of trunk
git fetch upstream

# Go to the feature branch
git checkout cool-feature

# Make a backup in case you mess up
git branch tmp cool-feature

# Rebase cool-feature onto trunk
git rebase --onto upstream/master upstream/master cool-feature
```

In this situation, where you are already on branch `cool-feature`, the last command can be written more succinctly as:

```
git rebase upstream/master
```

When all looks good you can delete your backup branch:

```
git branch -D tmp
```

If it doesn't look good you may need to have a look at *Recovering from mess-ups*.

If you have made changes to files that have also changed in trunk, this may generate merge conflicts that you need to resolve - see the git rebase man page for some instructions at the end of the "Description" section. There is some related help on merging in the git user manual - see resolving a merge.

If your feature branch is already on GitHub and you rebase, you will have to force push the branch; a normal push would give an error. If the branch you rebased is called `cool-feature` and your GitHub fork is available as the remote called `origin`, you use this command to force-push:

```
git push -f origin cool-feature
```

Note that this will overwrite the branch on GitHub, i.e. this is one of the few ways you can actually lose commits with git. Also note that it is never allowed to force push to the main astropy repo (typically called `upstream`), because this would re-write commit history and thus cause problems for all others.

Recovering from mess-ups

Sometimes, you mess up merges or rebases. Luckily, in git it is relatively straightforward to recover from such mistakes.

If you mess up during a rebase:

```
git rebase --abort
```

If you notice you messed up after the rebase:

```
# Reset branch back to the saved point
git reset --hard tmp
```

If you forgot to make a backup branch:

```
# Look at the reflog of the branch
git reflog show cool-feature

8630830 cool-feature@{0}: commit: BUG: io: close file handles immediately
278dd2a cool-feature@{1}: rebase finished: refs/heads/my-feature-branch onto 11ee694744f2552d
26aa21a cool-feature@{2}: commit: BUG: lib: make seek_gzip_factory not leak gzip obj
...

# Reset the branch to where it was before the botched rebase
git reset --hard cool-feature@{2}
```

Rewriting commit history

Note: Do this only for your own feature branches.

There's an embarrassing typo in a commit you made? Or perhaps the you made several false starts you would like the posterity not to see.

This can be done via *interactive rebasing*.

Suppose that the commit history looks like this:

```
git log --oneline
eadc391 Fix some remaining bugs
a815645 Modify it so that it works
2dec1ac Fix a few bugs + disable
13d7934 First implementation
6ad92e5 * masked is now an instance of a new object, MaskedConstant
29001ed Add pre-nep for a copule of structured_array_extensions.
...
```

and `6ad92e5` is the last commit in the `cool-feature` branch. Suppose we want to make the following changes:

- Rewrite the commit message for `13d7934` to something more sensible.
- Combine the commits `2dec1ac`, `a815645`, `eadc391` into a single one.

We do as follows:

```
# make a backup of the current state
git branch tmp HEAD
# interactive rebase
git rebase -i 6ad92e5
```

This will open an editor with the following text in it:

```
pick 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
pick a815645 Modify it so that it works
pick eadc391 Fix some remaining bugs

# Rebase 6ad92e5..eadc391 onto 6ad92e5
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

To achieve what we want, we will make the following changes to it:

```
r 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
f a815645 Modify it so that it works
f eadc391 Fix some remaining bugs
```

This means that (i) we want to edit the commit message for `13d7934`, and (ii) collapse the last three commits into one. Now we save and quit the editor.

Git will then immediately bring up an editor for editing the commit message. After revising it, we get the output:

```
[detached HEAD 721fc64] FOO: First implementation
 2 files changed, 199 insertions(+), 66 deletions(-)
[detached HEAD 0f22701] Fix a few bugs + disable
```

```
 1 files changed, 79 insertions(+), 61 deletions(-)
Successfully rebased and updated refs/heads/my-feature-branch.
```

and the history looks now like this:

```
0f22701 Fix a few bugs + disable
721fc64 ENH: Sophisticated feature
6ad92e5 * masked is now an instance of a new object, MaskedConstant
```

If it went wrong, recovery is again possible as explained *above*.

Converting a GitHub issue to a pull request

Sometimes you have a branch in your own GitHub repository designed to fix one particular issue. If that issue is listed on GitHub, a natural way to address it is to convert the issue to a pull request by attaching code with the fix to the issue. This can currently only be done using the GitHub API (there's no button or anything on the web site that does it, at least as of 2/6/2012). There are two options to do this:

- You can use the script at https://gist.github.com/1750715 which will do this for you automatically - just download the script and run it as a python command-line script, using the `python issue2pr.py --help` option to determine the precise usage.
- You can use the `hub` command-line utility provided here by GitHub. Once installed, you can attach a branch to a pull request by doing:

  ```
  hub pull-request -i <ID> -b astropy:master -h <USER>:<BRANCH>
  ```

  where `<ID>` is the ID of the issue, `<USER>` is the username, and `<BRANCH>` is the name of the branch you want to attach to the issue. For example:

  ```
  hub pull-request -i 42 -b astropy:master -h galahad:feature
  ```

  will attach the `feature` branch from `galahad`'s Astropy repository to issue 42.

  The `hub` command can do a lot more to interact with GitHub, so be sure to read their documentation. For example, you can fetch all branches of a repository for a given user by doing:

  ```
  hub fetch <USER>
  ```

3.3.2 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 *Workflow for Developers*.

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 *Overview* 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
```

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.

If for any reason developers do not wish to or cannot contribute via pull requests, they can submit a patch as described in `patches`.

CHAPTER

FOUR

CODING GUIDELINES

This section describes requirements and guidelines that should be followed both for the core package and for affiliated packages.

Note: Affiliated packages will only be considered for integration as a module in the core package once these guidelines have been followed.

4.1 Interface and Dependencies

- All code must be compatible with Python 2.6, 2.7, as well as 3.1 and later. All files should include the preamble:

  ```
  from __future__ import print_function, division
  ```

  and therefore use the `print()` function from Python 3. In addition, the new Python 3 formatting style should be used (i.e. `"{0:s}".format("spam")` instead of `"%s" % "spam"`), although when using positional arguments, the position should always be specified (i.e. `"{:s}"` is not compatible with Python 2.6). Astropy automatically runs the 2to3 tool on the source code, so in cases where syntax is different between Python 2 and 3, the Python 2 syntax should be used.
- The core package and affiliated packages should be importable with no dependencies other than components already in the Astropy core, the Python Standard Library, and NumPy 1.4 or later.
- The package should be importable from the source tree at build time. This means that, for example, if the package relies on C extensions that have yet to be built, the Python code is still importable, even if none of its functionality will work. One way to ensure this is to import the functions in the C extensions only within the functions/methods that require them (see next bullet point).
- 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.
- General utilities necessary for but not specific to the package or sub-package should be placed in the `packagename.utils`. These utilities will be moved to the `astropy.utils` module when the package is integrated into the core package. If a utility is already present in `astropy.utils`, the package should always use that utility instead of re-implementing it in `packagename.utils`. Note that the same applies to `astropy.tools`, which is intended for Astronomy-specific utilities.

4.2 Documentation and Testing

- Docstrings must be present for all public classes/methods/functions, and must follow the form outlined in the *Documentation Guidelines* document.
- 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.

4.3 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 `astropy.config.get_data_fileobj()` or `astropy.config.get_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 `astropy.config.get_data_fileobj()` or `astropy.config.get_data_filename()` functions. If a specific version of a data file is needed, the hash mechanism described in *Configuration system (astropy.config)* should be used.
- All persistent configuration should use the `astropy.config.ConfigurationItem` 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.

4.4 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)`. These get redirected to `log.warn` by default, but one can still use the standard warning-catching mechanism and custom warning classes.
- 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
```

4.5 Coding Style/Conventions

- The code will follow the standard PEP8 Style Guide for Python Code. In particular, this includes using only 4 spaces for indentation, and never tabs.
- One exception is to be made from the PEP8 style: new style relative imports of the form `from . import modname` are allowed and required for Astropy, as opposed to absolute (as PEP8 suggets) or the simpler `import modname` syntax. This is primarily due to improved relative import support since PEP8 was developed, and to simplify the process of moving modules.

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 `import numpy as np`, `import matplotlib as mpl`, and `import matplotlib.pyplot as plt` naming conventions should be used wherever relevant. `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 *Acceptable use of from module import **.
- 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. *Properties vs. get_/set_* below illustrates this guideline.
- All new classes should be new-style classes inheriting from `object` (in Python 3 this is a non-issue as all classes are new-style by default). The one exception to this rule is older classes in third-party libraries such the Python standard library or numpy.
- 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. *super() vs. Direct Calling* illustrates why this is important.
- Multiple inheritance should be avoided in general without good reason. Mulitple 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 simple be a single file, rather than a directory with an `__init__.py` file.
- When `try...except` blocks are used to catch exceptions, the `as` syntax should always be used, because this is available in all supported versions of python and is less ambiguous syntax (see *try...except block "as" syntax*).
- Command-line scripts should follow the form outlined in the *Writing Command-Line Scripts* document.

4.6 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, as per the example in the template package. Cython extensions should store `.pyx` files in the source code repository, but they should be compiled to `.c` files that are updated in the repository when important changes are made to the `.pyx` file.

- 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.
- 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*.

4.7 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 are required to follow the layout and documentation form of the template package included in the core package source distribution.
- 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. Recommended naming conventions for an affiliated package are either simply `packagename` or `awastropy.packagename` ("affiliated with Astropy").

4.8 Examples

This section shows a few examples (not all of which are correct!) to illustrate points from the guidelines. These will be moved into the template project once it has been written.

4.8.1 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
```

4.8.2 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 inherts 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(B, self).method()


class C(A):
    def method(self):
        print 'Doing C'
        super(C, self).method()

class D(C, B):
    def method(self):
        print 'Doing D'
        super(D, self).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 info on the pros and cons of using super, see http://rhettinger.wordpress.com/2011/05/26/super-considered-super/ or http://keithdevens.com/weblog/archive/2011/Mar/16/Python.super)

4.8.3 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()`.

4.8.4 try...except block "as" syntax

Catching of exceptions should always use this syntax:

```
try:
    ... some code that might produce a variety of exceptions ...
except ImportError as e:
    if 'somemodule' in e.args[0]"
        #for whatever reason, failed import of somemodule is ok
        pass
    else:
        raise
except ValueError, TypeError as e:
    msg = 'Hit an input problem, which is ok,'
    msg2 = 'but we're printing it here just so you know:'
    print msg, msg2, e
```

This avoids the old style syntax of `except ImportError, e` or `except (ValueError,TypeError), e`, which is dangerous because it's easy to instead accidentally do something like `except ValueError,TypeError`, which won't catch `TypeError`.

4.9 Additional Resources

Further tips and hints relating to the coding guidelines are included below.

4.9.1 Emacs setup for following coding guidelines

The Astropy coding guidelines are listed in *Coding Guidelines*. This document will describe some configuration options for Emacs, that will help in ensuring that Python code satisfies the guidelines. Emacs can be configured in several different ways. So instead of providing a drop in configuration file, only the individual configurations are presented below.

For this setup we will need flymake, pyflakes and the pep8 Python script, in addition to `python-mode`.

Flymake comes with Emacs 23. The rest can be obtained from their websites, or can be installed using `easy_install` or `pip`.

Global settings

No tabs

This setting will cause all tabs to be replaced with spaces. The number of spaces to use is set in the *Basic settings* section below.

```
;; Don't use TABS for indentations.
(setq-default indent-tabs-mode nil)
```

Maximum number of characters in a line

Emacs will automatically insert a new line after "fill-column" number of columns. PEP8 specifies a maximum of 79, but this can be set to a smaller value also, for example 72.

```
;; Set the number to the number of columns to use.
(setq-default fill-column 79)

;; Add Autofill mode to mode hooks.
(add-hook 'text-mode-hook 'turn-on-auto-fill)

;; Show line number in the mode line.
(line-number-mode 1)

;; Show column number in the mode line.
(column-number-mode 1)
```

Syntax highlighting

Enable syntax highlighting. This will also highlight lines that form a region.

```
(global-font-lock-mode 1)
```

Python specific settings

Basic settings

Indentation is automatically added. When a tab is pressed it is replaced with 4 spaces. When backspace is pressed on an empty line, the cursor will jump to the previous indentation level.

```
(load-library "python")

(autoload 'python-mode "python-mode" "Python Mode." t)
(add-to-list 'auto-mode-alist '("\\.py\\'" . python-mode))
(add-to-list 'interpreter-mode-alist '("python" . python-mode))

(setq interpreter-mode-alist
      (cons '("python" . python-mode)
            interpreter-mode-alist)
      python-mode-hook
      '(lambda () (progn
                    (set-variable 'py-indent-offset 4)
                    (set-variable 'indent-tabs-mode nil))))
```

Highlight the column where a line must stop

The "fill-column" column is highlighted in red. For this to work, download column-marker.el and place it in the Emacs configuration directory.

```
;; Highlight character at "fill-column" position.
(require 'column-marker)
(set-face-background 'column-marker-1 "red")
(add-hook 'python-mode-hook
          (lambda () (interactive)
            (column-marker-1 fill-column)))
```

Flymake

Flymake will mark lines that do not satisfy syntax requirements in red. When 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.

For flymake to work with `pep8` and `pyflakes`, create an executable file named `pychecker` with the following contents. This file must be in the system path.

```
#!/bin/bash

pyflakes "$1"
pep8 --ignore=E221,E701,E202 --repeat "$1"
true
```

Add the following code to Emacs configurations.

```
;; Setup for Flymake code checking.
(require 'flymake)
(load-library "flymake-cursor")
```

```
;; Script that flymake uses to check code. This script must be
;; present in the system path.
(setq pycodechecker "pychecker")

(when (load "flymake" t)
  (defun flymake-pycodecheck-init ()
    (let* ((temp-file (flymake-init-create-temp-buffer-copy
                       'flymake-create-temp-inplace))
           (local-file (file-relative-name
                        temp-file
                        (file-name-directory buffer-file-name))))
      (list pycodechecker (list local-file))))
  (add-to-list 'flymake-allowed-file-name-masks
               '("\\.py\\'" flymake-pycodecheck-init)))

(add-hook 'python-mode-hook 'flymake-mode)
```

Note: Flymake will save files with suffix *_flymake* in the current directory. If it crashes for some reason, then these files will not get deleted.

Sometimes there is a delay in refreshing the results.

Delete trailing white spaces and blank lines

To manually delete trailing whitespaces, press `C-t C-w`, which will run the command "delete-whitespaces'. This command is also run when a file is saved, and hence all trailing whitespaces will be deleted on saving a Python file.

To make sure that all "words" are separated by only one space, type `M-SPC` (use the ALT key since `M-SPC` sometimes brings up a context menu.).

To collapse a set of blank lines to one blank line, place the cursor on one of these and press `C-x C-o`. This is useful for deleting multiple black lines at the end of a file.

```
;; Remove trailing whitespace manually by typing C-t C-w.
(add-hook 'python-mode-hook
          (lambda ()
            (local-set-key (kbd "C-t C-w")
                           'delete-trailing-whitespace)))

;; Automatically remove trailing whitespace when file is saved.
(add-hook 'python-mode-hook
      (lambda()
        (add-hook 'local-write-file-hooks
              '(lambda()
                 (save-excursion
                   (delete-trailing-whitespace))))))

;; Use M-SPC (use ALT key) to make sure that words are separated by
;; just one space. Use C-x C-o to collapse a set of empty lines
;; around the cursor to one empty line. Useful for deleting all but
;; one blank line at end of file. To do this go to end of file (M->)
;; and type C-x C-o.
```

CHAPTER

FIVE

DOCUMENTATION GUIDELINES

5.1 Building the Documentation from source

For information about building the documentation from source, see the *Building documentation* section in the installation instructions.

5.2 Astropy Documentation Rules and Recommendations

This section describes the standards for documentation format affiliated packages that must follow for consideration of integration into the core module, as well as the standard Astropy docstring format.

- All documentation should be written use the Sphinx documentation tool.
- The template package will provide a recommended general structure for documentation.
- Docstrings must be provided for all public classes, methods, and functions.
- Docstrings will be incorporated into the documentation using a version of numpydoc included with Astropy, and should follow the NumPy/SciPy docstring standards, included below.
- Examples and/or tutorials are strongly encouraged for typical use-cases of a particular module or class.
- Any external package dependencies aside from NumPy, SciPy, or Matplotlib must be explicitly mentioned in the documentation.
- Configuration options using the `astropy.config` mechanisms must be explicitly mentioned in the documentation.

5.3 NumPy/SciPy Docstring Rules

The original source for these docstring standards is the NumPy project, and the associated numpydoc tools. The most up-to-date version of these standards can be found at numpy's github site. The guidelines below have been adapted to the Astropy pacakge.

5.3.1 Overview

In general, we follow the standard Python style conventions as described here:

- Style Guide for C Code

- Style Guide for Python Code
- Docstring Conventions

Additional PEPs of interest regarding documentation of code:

- Docstring Processing Framework
- Docutils Design Specification

Use a code checker:

- pylint
- pyflakes
- pep8.py

The following import conventions are used throughout the Astropy source and documentation:

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
```

Do not abbreviate `scipy`. There is no motivating use case to abbreviate it in the real world, so we avoid it in the documentation to avoid confusion.

It is not necessary to do `import numpy as np` at the beginning of an example. However, some sub-modules, such as `fft`, are not imported by default, and you have to include them explicitly:

```
import numpy.fft
```

after which you may use it:

```
np.fft.fft2(...)
```

5.3.2 Docstring Standard

A documentation string (docstring) is a string that describes a module, function, class, or method definition. The docstring is a special attribute of the object (`object.__doc__`) and, for consistency, is surrounded by triple double quotes, i.e.:

```
"""This is the form of a docstring.

It can be spread over several lines.

"""
```

NumPy and SciPy have defined a common convention for docstrings that provides for consistency, while also allowing our toolchain to produce well-formatted reference guides. This format should be used for Astropy docstrings.

This docstring standard uses re-structured text (reST) syntax and is rendered using Sphinx (a pre-processor that understands the particular documentation style we are using). While a rich set of markup is available, we limit ourselves to a very basic subset, in order to provide docstrings that are easy to read on text-only terminals.

A guiding principle is that human readers of the text are given precedence over contorting docstrings so our tools produce nice output. Rather than sacrificing the readability of the docstrings, we have written pre-processors to assist Sphinx in its task.

The length of docstring lines should be kept to 75 characters to facilitate reading the docstrings in text terminals.

5.3.3 Sections

The sections of the docstring are:

1. **Short summary**

   A one-line summary that does not use variable names or the function name, e.g.

   ```
   def add(a, b):
      """The sum of two numbers.

      """
   ```

   The function signature is normally found by introspection and displayed by the help function. For some functions (notably those written in C) the signature is not available, so we have to specify it as the first line of the docstring:

   ```
   """
   add(a, b)

   The sum of two numbers.

   """
   ```

2. **Deprecation warning**

   A section (use if applicable) to warn users that the object is deprecated. Section contents should include:

   - In what Astropy version the object was deprecated, and when it will be removed.
   - Reason for deprecation if this is useful information (e.g., object is superseded, duplicates functionality found elsewhere, etc.).
   - New recommended way of obtaining the same functionality.

   This section should use the note Sphinx directive instead of an underlined section header.

   ```
   .. note:: Deprecated in Astropy 1.2
             `ndobj_old` will be removed in Astropy 2.0, it is replaced by
             `ndobj_new` because the latter works also with array subclasses.
   ```

3. **Extended summary**

   A few sentences giving an extended description. This section should be used to clarify *functionality*, not to discuss implementation detail or background theory, which should rather be explored in the **notes** section below. You may refer to the parameters and the function name, but parameter descriptions still belong in the **parameters** section.

4. **Parameters**

   Description of the function arguments, keywords and their respective types.

   ```
   Parameters
   ----------
   x : type
      Description of parameter `x`.
   ```

Enclose variables in single backticks.

For the parameter types, be as precise as possible. Below are a few examples of parameters and their types.

```
Parameters
----------
filename : str
copy : bool
dtype : data-type
iterable : iterable object
shape : int or tuple of int
files : list of str
```

If it is not necessary to specify a keyword argument, use `optional`:

```
x : int, optional
```

Optional keyword parameters have default values, which are displayed as part of the function signature. They can also be detailed in the description:

```
Description of parameter `x` (the default is -1, which implies summation
over all axes).
```

When a parameter can only assume one of a fixed set of values, those values can be listed in braces:

```
order : {'C', 'F', 'A'}
    Description of `order`.
```

When two or more input parameters have exactly the same type, shape and description, they can be combined:

```
x1, x2 : array_like
    Input arrays, description of `x1`, `x2`.
```

5. **Returns**

   Explanation of the returned values and their types, of the same format as **parameters**.

6. **Other parameters**

   An optional section used to describe infrequently used parameters. It should only be used if a function has a large number of keyword parameters, to prevent cluttering the **parameters** section.

7. **Raises**

   An optional section detailing which errors get raised and under what conditions:

```
Raises
------
InvalidWCSException
    If the WCS information is invalid.
```

   This section should be used judiciously, i.e only for errors that are non-obvious or have a large chance of getting raised.

8. **See Also**

   An optional section used to refer to related code. This section can be very useful, but should be used judiciously. The goal is to direct users to other functions they may not be aware of, or have easy means of discovering (by looking at the module docstring, for example). Routines whose docstrings further explain parameters used by this function are good candidates.

   As an example, for a hypothetical function `astropy.wcs.world2pix` converting sky to pixel coordinates, we would have:

   ```
   See Also
   --------
   pix2world : Convert pixel to sky coordinates
   ```

   When referring to functions in the same sub-module, no prefix is needed, and the tree is searched upwards for a match.

   Prefix functions from other sub-modules appropriately. E.g., whilst documenting a hypothetical `astropy.vo` module, refer to a function in `table` by

   ```
   table.read : Read in a VO table
   ```

   When referring to an entirely different module:

   ```
   astropy.coords : Coordinate handling routines
   ```

   Functions may be listed without descriptions, and this is preferable if the functionality is clear from the function name:

   ```
   See Also
   --------
   func_a : Function a with its description.
   func_b, func_c_, func_d
   func_e
   ```

9. **Notes**

   An optional section that provides additional information about the code, possibly including a discussion of the algorithm. This section may include mathematical equations, written in LaTeX format:

   ```
   The FFT is a fast implementation of the discrete Fourier transform:

   .. math:: X(e^{j\omega } ) = x(n)e^{ - j\omega n}
   ```

   Equations can also be typeset underneath the math directive:

   ```
   The discrete-time Fourier time-convolution property states that

   .. math::

        x(n) * y(n) \Leftrightarrow X(e^{j\omega } )Y(e^{j\omega } )\\
        another equation here
   ```

   Math can furthermore be used inline, i.e.

```
The value of :math:‘\omega‘ is larger than 5.
```

Variable names are displayed in typewriter font, obtained by using `\mathtt{var}`:

```
We square the input parameter ‘alpha‘ to obtain
:math:‘\mathtt{alpha}^2‘.
```

Note that LaTeX is not particularly easy to read, so use equations sparingly.

Images are allowed, but should not be central to the explanation; users viewing the docstring as text must be able to comprehend its meaning without resorting to an image viewer. These additional illustrations are included using:

```
.. image:: filename
```

where filename is a path relative to the reference guide source directory.

10. **References**

    References cited in the **notes** section may be listed here, e.g. if you cited the article below using the text `[1]_`, include it as in the list as follows:

    ```
    .. [1] O. McNoleg, "The integration of GIS, remote sensing,
       expert systems and adaptive co-kriging for environmental habitat
       modelling of the Highland Haggis using object-oriented, fuzzy-logic
       and neural-network techniques," Computers & Geosciences, vol. 22,
       pp. 585-588, 1996.
    ```

    which renders as

    Referencing sources of a temporary nature, like web pages, is discouraged. References are meant to augment the docstring, but should not be required to understand it. References are numbered, starting from one, in the order in which they are cited.

11. **Examples**

    An optional section for examples, using the doctest format. This section is meant to illustrate usage, not to provide a testing framework – for that, use the `tests/` directory. While optional, this section is very strongly encouraged.

    When multiple examples are provided, they should be separated by blank lines. Comments explaining the examples should have blank lines both above and below them:

    ```
    >>> astropy.wcs.world2pix(233.2, -12.3)
    (134.5, 233.1)

    Comment explaining the second example

    >>> astropy.coords.fk5_to_gal("00:42:44.33 +41:16:07.5")
    (121.1743, -21.5733)
    ```

    For tests with a result that is random or platform-dependent, mark the output as such:

    ```
    >>> astropy.coords.randomize_position(244.9, 44.2, radius=0.1)
    (244.855, 44.13)  # random
    ```

It is not necessary to use the doctest markup <BLANKLINE> to indicate empty lines in the output. The examples may assume that `import numpy as np` is executed before the example code.

5.3.4 Documenting classes

Class docstrings

Use the same sections as outlined above (all except `Returns` are applicable). The constructor (`__init__`) should also be documented here, the `Parameters` section of the docstring details the constructors parameters.

An `Attributes` section, located below the `Parameters` section, may be used to describe class variables:

```
Attributes
----------
x : float
    The X coordinate.
y : float
    The Y coordinate.
```

Attributes that are properties and have their own docstrings can be simply listed by name:

```
Attributes
----------
real
imag
x : float
    The X coordinate
y : float
    The Y coordinate
```

In general, it is not necessary to list class methods. Those that are not part of the public API have names that start with an underscore. In some cases, however, a class may have a great many methods, of which only a few are relevant (e.g., subclasses of ndarray). Then, it becomes useful to have an additional `Methods` section:

```
class Table(ndarray):
    """
    A class to represent tables of data

    ...

    Attributes
    ----------
    columns : list
        List of columns

    Methods
    -------
    read(filename)
        Read a table from a file
    sort(column, order='ascending')
        Sort by `column`
    """
```

If it is necessary to explain a private method (use with care!), it can be referred to in the **extended summary** or the **notes**. Do not list private methods in the Methods section.

Do not list `self` as the first parameter of a method.

Method docstrings

Document these as you would any other function. Do not include `self` in the list of parameters. If a method has an equivalent function, the function docstring should contain the detailed documentation, and the method docstring should refer to it. Only put brief `Summary` and `See Also` sections in the method docstring.

5.3.5 Documenting class instances

Instances of classes that are part of the Astropy API may require some care. To give these instances a useful docstring, we do the following:

- Single instance: If only a single instance of a class is exposed, document the class. Examples can use the instance name.
- Multiple instances: If multiple instances are exposed, docstrings for each instance are written and assigned to the instances' `__doc__` attributes at run time. The class is documented as usual, and the exposed instances can be mentioned in the Notes and See Also sections.

5.3.6 Documenting constants

Use the same sections as outlined for functions where applicable:

```
1. summary
2. extended summary (optional)
3. see also (optional)
4. references (optional)
5. examples (optional)
```

Docstrings for constants will not be visible in text terminals (constants are of immutable type, so docstrings can not be assigned to them like for for class instances), but will appear in the documentation built with Sphinx.

5.3.7 Documenting modules

Each module should have a docstring with at least a summary line. Other sections are optional, and should be used in the same order as for documenting functions when they are appropriate:

```
1. summary
2. extended summary
3. routine listings
4. see also
5. notes
6. references
7. examples
```

Routine listings are encouraged, especially for large modules, for which it is hard to get a good overview of all functionality provided by looking at the source file(s) or the `__all__` dict.

Note that license and author info, while often included in source files, do not belong in docstrings.

5.3.8 Other points to keep in mind

- Notes and Warnings : If there are points in the docstring that deserve special emphasis, the reST directives for a note or warning can be used in the vicinity of the context of the warning (inside a section). Syntax:

```
.. warning:: Warning text.

.. note:: Note text.
```

Use these sparingly, as they do not look very good in text terminals and are not often necessary. One situation in which a warning can be useful is for marking a known bug that is not yet fixed.

- Questions and Answers : For general questions on how to write docstrings that are not answered in this document, refer to http://docs.scipy.org/numpy/Questions+Answers/.
- `array_like` : For functions that take arguments which can have not only a type `ndarray`, but also types that can be converted to an ndarray (i.e. scalar types, sequence types), those arguments can be documented with type `array_like`.

5.3.9 Common reST concepts

For paragraphs, indentation is significant and indicates indentation in the output. New paragraphs are marked with a blank line.

Use *italics*, **bold**, and `courier` if needed in any explanations (but not for variable names and doctest code or multi-line code). Variable, module and class names should be written between single back-ticks (`‘astropy‘`).

A more extensive example of reST markup can be found in this example document; the quick reference is useful while editing.

Line spacing and indentation are significant and should be carefully followed.

5.3.10 Conclusion

An example of the format shown here is available. Refer to How to Build API/Reference Documentation on how to use Sphinx to build the manual.

5.4 Sphinx Documentation Themes

A custom Sphinx HTML theme is included in the astropy source tree and installed along with astropy. This allows the theme to be used by default from both astropy and affiliated packages. This is done by setting the theme in the global astropy sphinx configuration, which is imported in the sphinx configuration of both `astropy` and affiliated packages.

5.4.1 Using a different theme for `astropy` or 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.

5.4.2 Adding more custom themes to astropy

Additional custom themes can be included in the astropy source tree by placing them in the directory `astropy/astropy/sphinx/themes`, and editing `astropy/astropy/sphinx/setup_package.py` to include the theme (so that it is installed).

CHAPTER

SIX

TESTING GUIDELINES

This section describes the testing framework and format standards for tests in Astropy core packages (this also serves as recommendations for affiliated packages).

6.1 Testing Framework

The testing framework used by Astropy is the py.test framework.

6.2 Running Tests

There are currently three different ways to invoke Astropy tests. Each method invokes py.test to run the tests but offers different options when calling.

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.

6.2.1 setup.py test

The safest way to run the astropy test suite is via the setup command `test`. This is invoked by running `python setup.py test` while in the astropy source code directory. Run `python setup.py test --help` to see the options to the test command.

Turn on PEP8 checking by passing `--pep8` to the `test` command. This will turn off regular testing and enable PEP8 testing.

Note: This method of running the tests defaults to the version of `py.test` that is bundled with Astropy. To use the locally-installed version, you can set the `ASTROPY_USE_SYSTEM_PYTEST` environment variable, eg.:

```
> ASTROPY_USE_SYSTEM_PYTEST=1 python setup.py test
```

6.2.2 py.test

An alternative way to run tests from the command line is to switch to the source code directory of astropy and simply type:

```
py.test
```

`py.test` will look for files that look like tests in the currect directory and all recursive directories then run all the code that looks like tests within those files.

Note: To test any compiled C/Cython extensions, you must run `python setup.py develop` prior to running the py.test command-line script. Otherwise, any tests that make use of these extensions will not succeed. Similarly, in python 3, these tests will not run correctly in the source code, because they need the `2to3` tool to be run on them.

You may specify a specific test file or directory at the command line:

```
py.test test_file.py
```

To run a specific test within a file use the `-k` option:

```
py.test test_file.py -k "test_function"
```

You may also use the `-k` option to not run tests py putting a `-` in front of the matching string:

```
py.test test_file.py -k "-test_function"
```

py.test has a number of command line usage options.

Turn on PEP8 testing by adding the `--pep8` flag to the `py.test` call. By default regular tests will also be run but these can be turned off by adding `-k pep8`:

```
py.test some_dir --pep8 -k pep8
```

Note: This method of running the tests uses the locally-installed version of `py.test` rather than the bundled one, and hence will fail if the local version it is not up-to-date enough (`py.test` 2.2 as of this writing).

6.2.3 astropy.test()

AstroPy includes a standalone version of py.test that allows to tests to be run even if py.test is not installed. Tests can be run from within AstroPy with:

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

This will run all the default tests for AstroPy.

Tests for a specific package can be run by specifying the package in the call to the `test()` function:

```
astropy.test('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('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=, pastebin= and coverage=.

Enable PEP8 compliance testing with pep8=True in the call to astropy.test. This will enable PEP8 checking and disable regular tests.

Note: This method of running the tests defaults to the version of py.test that is bundled with Astropy. To use the locally-installed version, you should set the ASTROPY_USE_SYSTEM_PYTEST environment variable (see *Configuration system (astropy.config)*) or the py.test method described above.

6.3 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.

6.4 Where to put tests

6.4.1 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, either in a tests directory, or in a test.py file, e.g:

```
astropy/io/fits/tests/
```

or:

```
astropy/io/fits/test.py
```

tests directories should contain an __init__.py file so that the tests can be imported and so that they can use relative imports.

6.4.2 Interoperability tests

Tests involving two or more sub-packages should be included in:

```
astropy/tests/
```

and using:

```
astropy.test()
```

then runs both these interoperability tests, and all the unit tests in the sub-packages. This functionality is especially important for people who install packages through bundles and package managers, where the original source code for the tests is not immediately available.

6.5 Writing tests

`py.test` 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 `py.test`.

6.5.1 Simple example

The following example shows a simple function and a test to test this function:

```
def func(x):
    return x + 1

def test_answer():
    assert func(3) == 5
```

If we place this in a `test.py` file and then run:

```
py.test test.py
```

The result is:

```
============================= test session starts ==============================
python: platform darwin -- Python 2.7.2 -- pytest-1.1.1
test object 1: /Users/tom/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 ============================
```

6.5.2 Working with data files

Tests that need to make use of a data file should use the `get_data_fileobj` or `get_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.

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 `@remote_data` decorator. Tests marked with this decorator 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=True` flag. Turn on the remote data tests at the command line with `py.test --remote-data`.

Examples

```
from ...config import get_data_filename
from ...tests.helper import remote_data

def test_1():
    #if filename.fits is a local file in the source distribution
    datafile = get_data_filename('filename.fits')
    # do the test

@remote_data
def test_2():
    #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
```

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.

6.5.3 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 py.test tmpdir function argument or with Python's built-in tempfile module.

6.5.4 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):
    module.NUM = 11

def add_num(x):
```

```
    return x + NUM

def test_42():
    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):
    module.DATAFILE = get_remote_test_data('94935ac31d585f68041c08f87d1a19d4')

def test():
    f = open(DATAFILE, 'rb')
    # do the test

def teardown_module(module):
    os.remove(DATAFILE)
```

Class-level

Tests can be organized into classes that have their own setup/teardown functions. In the following

```
def add_nums(x, y):
    return x + y

class TestAdd42(object):

    def setup_class(self):
        self.NUM = 42

    def test_1(self):
        added = add_nums(11, self.NUM)
        assert added == 53

    def test_2(self):
        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

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):
    return x + y

class TestAdd42(object):
```

```
    def setup_method(self, method):
        self.NUM = 42

    def test_1(self):
        added = add_nums(11, self.NUM)
        assert added == 53

    def test_2(self):
        added = add_nums(13, self.NUM)
        assert added == 55

    def teardown_method(self, method):
        pass
```

Function-level

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):
    # do test

def test_2(self):
    # do test

def teardown_method(function):
    pass
```

6.5.5 Parametrizing tests

If you want to run a test several times for slightly different values, then it can be advantageous to use the `py.test` option to parametrize 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 `parametrize` decorator to loop over the different inputs:

```
@pytest.mark.parametrize(('letter'), ['a', 'b', 'c'])
def test(letter):
    assert type(letter) == str
```

6.5.6 Using py.test helper functions

If your tests need to use py.test helper functions, such as `pytest.raises`, import `pytest` into your test module like so:

```
from ...tests.helper import pytest
```

You may need to adjust the relative import to work for the depth of your module. `tests.helper` imports `pytest` either from the user's system or `extern.pytest` if the user does not have py.test installed. This is so that users need not install py.test to run AstroPy's tests.

6.6 Using data in tests

Tests can include very small datafiles, but any files significantly larger than the source code should be placed on a remote server. The base URL for the test files will be:

```
http://data.astropy.org/
```

and files will be accessed by their MD5 hash, for example:

```
http://data.astropy.org/94935ac31d585f68041c08f87d1a19d4
```

Tests then retrieve data via this URL. This implicitly allows versioning, since different versions of data files will have different hashes. Old data files should not be removed, so that tests can be run in any version of AstroPy.

The details of the server implementation have yet to be decided, but using these static hash-based URLs ensures that even if we change the backend, the URL will remain the same.

6.7 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.

6.8 Test coverage reports

Astropy can use coverage.py to generate test coverage reports. To generate a test coverage report, use:

```
python setup.py test --coverage
```

There is a coveragerc file that defines files to omit as well as lines to exclude. It is installed along with astropy so that the `astropy.test` function can use it. In the source tree, it is at `astropy/tests/coveragerc`.

6.8.1 Marking blocks of code to exclude from coverage

Blocks of code may be ignored 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()
```

Blocks of code that are intended to run only in Python 2.x or 3.x may also be marked so that they will be ignored when appropriate by `coverage.py`:

```
if sys.version_info[0] >= 3:  # pragma: py3
    do_it_the_python3_way()
else:  # pragma: py2
    do_it_the_python2_way()
```

CHAPTER

SEVEN

BUILDING, CYTHON/C EXTENSIONS, AND RELEASING

The build process currently uses the Distribute package to build and install the astropy core (and any affiliated packages that use the template). The user doesn't necessarily need to have `distribute` installed, as it will automatically bootstrap itself using the `distribute_setup.py` file in the source distribution if it isn't installed for the user.

7.1 Customizing setup/build for subpackages

As is typical, there is a single `setup.py` file that is used for the whole `astropy` package. To customize setup parameters for a given sub-package, a `setup_package.py` file can be defined inside a package, and if it is present, the setup process will look for the following functions to customize the build process:

- `get_package_data()`
  This function, if defined, should return a dictionary mapping the name of the subpackage(s) that need package data to a list of data file paths (possibly including wildcards) relative to the path of the package's source code. e.g. if the source distribution has a needed data file `astropy/wcs/tests/data/3d_cd.hdr`, this function should return `{'astropy.wcs.tests:'['data/3d_cd.hdr']}`. See the `package_data` option of the `distutils.core.setup()` function.

  It is recommended that all such data be in a directory named "data" inside the package within which it is supposed to be used, and package data should be accessed via the `astropy.utils.data.get_data_filename` and `astropy.utils.data.get_data_fileobj` functions.

- `get_extensions()`
  This provides information for building C or Cython extensions. If defined, it should return a list of `distutils.core.Extension` objects controlling the Cython/C build process (see below for more detail).

- `get_legacy_alias()`
  This function allows for the creation of `shims` that allow a subpackage to be imported under another name. For example, `astropy.io.fits` used to be available under the namespace `pyfits`. For backward compatibility, it is helpful to have it still importable under the old name. Under most circumstances, this function should call `astropy.setup_helpers.add_legacy_alias` to generate a legacy module and then return what it returns.

- `get_build_options()`
  This function allows a package to add extra build options. It should return a list of tuples, where each element has:

  - *name*: The name of the option as it would appear on the commandline or in the `setup.cfg` file.
  - *doc*: A short doc string for the option, displayed by `setup.py build --help`.

  - *is_bool* (optional): When `True`, the option is a boolean option and doesn't have an associated value.

  Once an option has been added, its value can be looked up using `astropy.setup_helpers.get_distutils_build_option`.

- `get_external_libraries()`
  This function declares that the package uses libraries that are included in the astropy distribution that may also be distributed elsewhere on the users system. It should return a list of library names. For each library, a new build option is created, `--use-system-X` which allows the user to request to use the system's copy of the library. The package would typically call `astropy.setup_helpers.use_system_library` from its `get_extensions` function to determine if the package should use the system library or the included one.

The `astropy.setup_helpers` modules includes a `update_package_files()` function which automatically searches the given source path for `setup_package.py` modules and calls each of the above functions, if they exist. This makes it easy for affiliated packages to use this machinery in their own `setup.py`.

7.2 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 `distutils.core.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 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 extensions automatically generated extensions. Note that this allows disabling a Cython file by providing an extension that includes the Cython file, but giving it the special `name` 'cython_skip'. Any extension with this package name will not be built by `setup.py`.

Note: If an `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.

7.2.1 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 `get_package_data()` hook in `setup_package.py` to install those header files. For example, the `astropy.wcs` package has this:

```
def get_package_data():
    return {'astropy.wcs': ['include/*.h']}
```

7.3 Preventing importing at build time

In rare cases, some packages may need to be imported at build time. Unfortunately, anything that requires a C or Cython extension or processing through 2to3 will fail to import until the build phase has completed. In those cases, the `_ASTROPY_SETUP_` variable can be used to determine if the package is being imported as part of the build and choose to not import problematic modules. `_ASTROPY_SETUP_` is inserted into the builtins, and is `True` when inside of astropy's `setup.py` script, and `False` otherwise.

For example, suppose there is a subpackage `foo` that needs to import a module called `version.py` at build time in order to set some version information, and also has a C extension, `process`, that will not be available in the source tree. In this case, `astropy/foo/__init__.py` would probably want to check the value of `_ASTROPY_SETUP_` before importing the C extension:

```
if not _ASTROPY_SETUP_:
    from . import process

from . import version
```

7.4 Release

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.

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.

7.4.1 Creating a GPG Signing Key and a Signed Tag

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, simply 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
```

7.4.2 Release Procedure

The automated portion of the Astropy release procedure uses zest.releaser to create the tag and update the version. zest.releaser is extendable through hook functions–Astropy already includes a couple hook functions to modify the default behavior, but future releases may be further automated through the implementation of additional hook functions. In order to use the hooks, Astropy itself must be *installed* alongside zest.releaser. It is recommended to create a virtualenv specifically for this purpose.

This may seem like a lot of steps, but most of them won't be necessary to repeat for each release. The advantage of using an automated or semi-automated procedure is that ensures a consistent release process each time.

1. Update the list of contributors in the `creditsandlicense.rst` file. The easiest way to check this is do:

   ```
   $ git shortlog -s
   ```

   And just add anyone from that list who isn't already credited.

2. Install virtualenv if you don't already have it. See the linked virtualenv documentation for details. Also, make sure that you have cython installed, as you will need it to generate the .c files needed for the release.

3. Create and activate a virtualenv:

   ```
   $ virtualenv --system-site-packages --distribute astropy-release
   $ source astropy-release/bin/activate
   ```

4. Obtain a *clean* version of the Astropy repository. That is, one where you don't have any intermediate build files. Either use a fresh `git clone` or do `git clean -dfx`.

5. Be sure you're the "master" branch, and install Astropy into the virtualenv:

   ```
   $ python setup.py install
   ```

   This is necessary for two reasons. First, the entry points for the releaser scripts need to be availale, and these are in the Astropy package. Second, the build process will generate .c files from the Cython .pyx files, and the .c files are necessary for the source distribution.

6. Install zest.releaser into the virtualenv; use `--upgrade --force` to ensure that the latest version is installed in the virtualenv (if you're running a csh variant make sure to run `rehash` afterwards too):

```
$ pip install zest.releaser --upgrade --force
```

7. Ensure that all changes to the code have been committed, then start the release by running:

   ```
   $ fullrelease
   ```

8. You will be asked to enter the version to be released. Press enter to accept the default (which will normally be correct) or enter a specific version string. A diff will then be shown of CHANGES.rst and setup.py showing that a release date has been added to the changelog, and that the version has been updated in setup.py. Enter 'Y' when asked to commit these changes.
9. You will then be shown the command that will be run to tag the release. Enter 'Y' to confirm and run the command.
10. When asked "Check out the tag (for tweaks or pypi/distutils server upload)" enter 'N': zest.releaser does not offer enough control yet over how the register and upload are performed so we will do this manually until the release scripts have been improved.
11. You will be asked to enter a new development version. Normally the next logical version will be selected–press enter to accept the default, or enter a specific version string. Do not add ".dev" to the version, as this will be appended automatically (ignore the message that says ".dev0 will be appended"–it will actually be ".dev" without the 0). For example, if the just-released version was "0.1" the default next version will be "0.2". If we want the next version to be, say "1.0" then that must be entered manually.
12. You will be shown a diff of CHANGES.rst showing that a new section has been added for the new development version, and showing that the version has been updated in setup.py. Enter 'Y' to commit these changes.
13. When asked to push the changes to a remote repository, enter 'Y'. This should complete the portion of the process that's automated at this point.
14. Check out the tag of the released version. For example:

    ```
    $ git checkout v0.1
    ```

15. Create the source distribution by doing:

    ```
    $ python setup.py sdist
    ```

    Copy the produced `.tar.gz` somewhere and verify that you can unpack it, build it, and get all the tests to pass. It would be best to create a new virtualenv in which to do this.
16. Register the release on PyPI with:

    ```
    $ python setup.py register
    ```

17. Upload the source distribution to PyPI; this is preceded by re-running the sdist command, which is necessary for the upload command to know which distribution to upload:

    ```
    $ python setup.py sdist upload
    ```

18. Update the website to reflect the fact there is now a stable release.
19. Update Readthedocs so that it builds docs for the corresponding github tag, and set the default page to the new release.
20. Create a bug fix branch. If the version just was released was a "X.Y.0" version ("0.1" or "0.2" for example–the final ".0" is typically ommitted) it is good to create a bug fix branch as well. Starting from the tagged changset, just checkout a new branch and push it to the remote server. For example, after releasing version 0.1, do:

```
$ git checkout -b v0.1.x
```

Then edit `setup.py` so that the version is `'0.1.1.dev'`, and commit that change. Then, do:

```
$ git push upstream v0.1.x
```

Note:

You may need to replace `upstream` here with `astropy` or whatever remote name you use for the main astropy repository.

The purpose of this branch is for creating bug fix releases like "0.1.1" and "0.1.2", while allowing development of new features to continue in the master branch. Only changesets that fix bugs without making significant API changes should be merged to the bug fix branches.

---

21. Create a bug fix label on GitHub; this should have the same name as the just created bug fix branch. This label should be applied to all issues that should be backported to the bug fix branch.

7.4.3 Creating a MacOS X Installer on a DMG

The `bdist_dmg` command can be used to create a `.dmg` disk image for MacOS X with a `.pkg` installer. In order to do this, you will need to ensure that you have the following dependencies installed:

- Numpy
- Sphinx
- bdist_mpkg

To create a `.dmg` file, run:

```
python setup.py bdist_dmg
```

Note that for the actual release version, you should do this with the Python distribution from python.org (not e.g. MacPorts, EPD, etc.). The best way to ensure maximum compatibility is to make sure that Python and Numpy are installed into `/Library/Frameworks/Python.framework` using the latest stable `.dmg` installers available for those packages. In addition, the `.dmg` should be build on a MacOS 10.6 system, to ensure compatibility with 10.6, 10.7, and 10.8.

Before distributing, you should test out an installation of Python, Numpy, and Astropy from scratch using the `.dmg` installers, preferably on a clean virtual machine.

7.5 Future directions

We plan to switch to a newer packaging scheme when it's more stable, the upcoming standard library `packaging` module, derived from the distutils2 project. Until it's working right, however, we will be using `distribute` and `distutils`.

CHAPTER

EIGHT

WRITING COMMAND-LINE SCRIPTS

Command-line scripts in Astropy should follow a consistent scheme to promote readability and compatibility.

The actual script should be in the `/scripts` directory of the Astropy source distribution, and should do nothing aside from importing a `main` function from astropy and execute it. This is partly necessary because the "2to3" utility that converts python 2.x code to 3.x does not convert scripts. These scripts should be executable, include `#!/usr/bin/env python` at the top, and should *not* end in `.py`.

The `main` functions these scripts call should accept an optional single argument that holds the `sys.argv` list, except for the script name (e.g., `argv[1:]`). This function can live in its own module, or be part of a larger module that implements a class or function for astropy library use. The `main` function should do very little actual work - it should only parse the arguments and pass those arguments on to some library function so that the library function can be used programmatically when needed. 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`. `argparse` is only available in python >=2.7 and >=3.2, however, so it should be imported as `from astropy.util.compat import argparse` .

8.1 Example

Contents of `/scripts/cmdlinescript`

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""An astropy command-line script"""

import astropy.somepackage.somemod

astropy.somepackage.somemod.main()
```

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):
    from astropy.utils.compat 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)
```

CHAPTER

NINE

SPHINX EXTENSIONS

This page gives the API for the custom Sphinx extensions used in Astropy.

9.1 automodapi Extension

This sphinx extension adds a tools to simplify generating the API documentationfor Astropy packages and affiliated packages.

9.1.1 `automodapi` directive

This directive takes a single argument that must be module or package. Itwill produce a Documentation section named "Reference/API" that includes the docstring for the package, an `automodsumm` directive, and an `automod-diagram` if there are any classes in the module.

It accepts the following options:

- `:no-inheritance-diagram:`
  If present, the inheritance diagram will not be shown even if the module/package has classes.
- `:skip: str`
  This option results in the specified object being skipped, that is the object will *not* be included in the generated documentation. This option may appear any number of times to skip multiple objects.
- `:no-main-docstr:`
  If present, the docstring for the module/package will not be generated. The function and class tables will still be used, however.
- `:headings: str`
  Specifies the characters (in one string) used as the heading levels used for the generated section. This must have at least 2 characters (any after 2 will be ignored). This also *must* match the rest of the documentation on this page for sphinx to be happy. Defaults to "-^", which matches the convention used for Python's documentation, assuming the automodapi call is inside a top-level section (which usually uses '=').

This extension also adds a sphinx configuration option `automodapi_toctreedirnm`. It must be a string that specifies the name of the directory the automodsumm generated documentation ends up in. This directory path should be relative to the documentation root (e.g., same place as `index.rst`). It defaults to '_generated'

9.2 automodsumm Extension

This sphinx extension adds two directives for summarizing the public members of a module or package.

These directives are primarily for use with the automodapi extension, but can be used independently.

9.2.1 automodsumm directive

This directive will produce an "autosummary"-style table for public attributes of a specified module. See the sphinx.ext.autosummary extension for details on this process. The main difference from the autosummary directive is that autosummary requires manually inputting all attributes that appear in the table, while this captures the entries automatically.

This directive requires a single argument that must be a module or package.

It also accepts any options supported by the autosummary directive- see sphinx.ext.autosummary for details. It also accepts two additional options:

- :classes-only:
  If present, the autosummary table will only contain entries for classes. This cannot be used at the same time with :functions-only: .
- :functions-only:
  If present, the autosummary table will only contain entries for functions. This cannot be used at the same time with :classes-only: .
- :skip: obj1, [obj2, obj3, ...]
  If present, specifies that the listed objects should be skipped and not have their documentation generated, nor be includded in the summary table.

9.2.2 automod-diagram directive

This directive will produce an inheritance diagram like that of the sphinx.ext.inheritance_diagram extension.

This directive requires a single argument that must be a module or package. It accepts no options.

Note: Like 'inheritance-diagram', 'automod-diagram' requires graphviz to generate the inheritance diagram.

9.3 Numpydoc Extension

This extension is a port of the numpydoc extentension written by the numpy and scipy projects, with some tweaks for Astropy. See the code for details, as it is quite complex and includes a variety of interrelated extensions.

CHAPTER

TEN

FULL CHANGELOG

10.1 0.2 (unreleased)

- Nothing changed yet.

10.2 0.2b1 (2012-12-24)

10.2.1 New Features

This is a brief overview of the new features included in Astropy 0.2–please see the "What's New" section of the documentation for more details.

- `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]
- `astropy.table` I/O infrastructure for custom readers/writers implemented. [#305]
  - Added support for reading/writing HDF5 files [#461]
- 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.
- Added support for masked tables with missing or invalid data [#451]
- `astropy.wcs`
  - From updating the 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".

10.2.2 Other Changes and Additions

- 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.
- 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]

10.2.3 Bug Fixes

- `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 Paper IV 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]

10.3 0.1 (2012-06-19)

- Initial release.

Part V

Indices and Tables

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

a

PYTHON MODULE INDEX

a