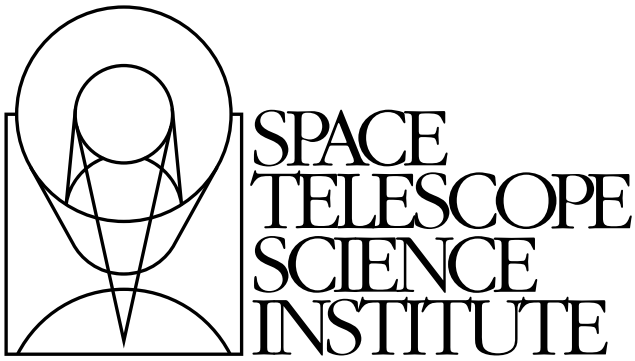


PyFITS Documentation

Release 3.1.7.dev

J. C. Hsu **Paul Barrett** **Christopher Hanley**
James Taylor **Michael Droettboom** **Erik M. Bray**

May 14, 2014

Contents

CHAPTER 1

PyFITS Users Guide

1.1 Introduction

The PyFITS module is a Python library providing 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.

1.1.1 Installation

PyFITS requires Python version 2.5 or newer. PyFITS also requires the numpy package. Information about numpy can be found at:

http://numpy.scipy.org/

To download numpy, go to:

http://sourceforge.net/project/numpy

PyFITS' source code is mostly pure Python, but includes an optional C module which wraps CFITSIO for compression support. The latest source distributions and binary installers for Windows can be downloaded from:

http://www.stsci.edu/resources/software_hardware/pyfits/Download

Or from the Python Package Index (PyPI) at:

https://pypi.python.org/pypi/pyfits

PyFITS uses Python's distutils for its installation. To install it, unpack the tar file and type:

```
python setup.py install
```

This will install PyFITS in the system's Python site-packages directory. If your system permissions do not allow this kind of installation, use of virtualenv for personal installations is recommended.

In this guide, we'll assume that the reader has basic familiarity with Python. Familiarity with numpy is not required, but it will help to understand the data structures in PyFITS.

1.1.2 User Support

The official PyFITS web page is:

http://www.stsci.edu/resources/software_hardware/pyfits

If you have any question or comment regarding PyFITS, user support is available through the STScI Help Desk:

* **E-mail:** help@stsci.edu
* **Phone:** (410) 338-1082

1.2 Quick Tutorial

This chapter provides a quick introduction of using PyFITS. The goal is to demonstrate PyFITS' basic features without getting into too much detail. If you are a first time user or an occasional PyFITS user, using only the most basic functionality, this is where you should start. Otherwise, it is safe to skip this chapter.

After installing numpy and PyFITS, start Python and load the PyFITS library. Note that the module name is all lower case.

```
>>> import pyfits
```

1.2.1 Reading and Updating Existing FITS Files

Opening a FITS file

Once the PyFITS module is loaded, we can open an existing FITS file:

```
>>> hdulist = pyfits.open('input.fits')
```

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 method returns a PyFITS object called an `HDUList` which is a `list`-like object, consisting of HDU objects. An HDU (Header Data Unit) is the highest level component of the FITS file structure, consisting of a header and (typically) a data array or table.

After the above open call, `hdulist[0]` is the primary HDU, `hdulist[1]` is the first extension HDU (if there are any extensions), and so. It should be noted that PyFITS 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 `pyfits.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. `memmap=True` is the default value as of PyFITS v3.1.0.

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.

Warning: When opening a file with `memmap=True`, because of how mmap works this means that when the HDU data is accessed (i.e. `hdul[0].data`) another handle to the FITS file is opened by mmap. This means that even after calling `hdul.close()` the mmap still holds an open handle to the data so that it can still be accessed by unwary programs that were built with the assumption that the `.data` attribute has all the data in-memory.
In order to force the mmap to close either wait for the containing `HDUList` object to go out of scope, or manually call `del hdul[0].data` (this works so long as there are no other references held to the data array).

Working with FITS Headers

As mentioned earlier, each element of an `HDUList` is an HDU object with `.header` and `.data` attributes, which can be used to access the header and data portions of the HDU.

For those unfamiliar with FITS headers, they consist of a list of 80 byte "cards", where a card contains a keyword, a value, and a comment. The keyword and comment must both be strings, whereas the value can be a string or an integer, floating point number, complex number, or `True`/`False`. Keywords are usually unique within a header, except in a few special cases.

The header attribute is a Header instance, another PyFITS 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 PyFITS 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 PyFITS, either through keyword name or index:

```
>>> prihdr = hdulist[0].header
>>> prihdr['targname'] = 'NGC121-a'
>>> prihdr[27] = 99
```

Please note however that almost all application code should update header values via their keyword name and not via their positional index. This is because most FITS keywords may appear at any position in the header.

It is also possible to update both the value and comment associated with a keyword by assigning them as a tuple:

```
>>> 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['history']
I updated this file 2/26/09
>>> 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 2/27/09'
>>> prihdr['history']
I updated this file on 2/27/09
>>> prihdr['comment'][1] = 'I like using JWST observations'
>>> prihdr['comment']
Edwin Hubble really knew his stuff
I like using JWST observations
```

To see the entire header as it appears in the FITS file (with the END card and padding stripped), 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...
```

Entering simply `print header` will also work, but may not be very legible on most displays, as this displays the header as it is written in the FITS file itself, which means there are no linebreaks between cards. This is a common confusion in new users of PyFITS.

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) 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, the EXTVER value needs to be specified along with the EXTNAME as a tuple; e.g.:

```
>>> scidata = hdulist['sci', 2].data
```

Note that the EXTNAME is also case-insensitive.

The returned numpy object has many attributes and methods for a user to get information about the array; 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 fastest changing 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
```

Note that performing an operation like this on an entire image requires holding the entire image in memory. This example performs the multiplication in-place so that no copies are made, but the original image must first be able to fit in main memory. For most observations this should not be an issue on modern personal computers.

If at this point you want to preserve all 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 numpy `recarray` (record array) objects, you will find the table data is basically a record array with some extra properties. But familiarity with record arrays is not a prerequisite for this guide.

Like images, the data portion of a FITS table extension is in the `.data` attribute:

```
>>> hdulist = pyfits.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_record` object which looks like a (Python) tuple containing elements of heterogeneous data types. In this example: an integer, a string, a floating point number, and a Boolean value. So the table data are just an array of such records. More commonly, a user is likely to access the data in a column-wise way. This is accomplished by using the `field()` method. To get the first column (or "field" in Numpy parlance–it is used here interchangeably with "column") of the table, use:

```
>>> 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])
```

When accessing a column by name, dict-like access is also possible (and even preferable):

```
>>> tbdata['id']
array([1, 2])
```

In most cases it is preferable to access columns by their name, as the column name is entirely independent of its physical order in the table. As with header keywords, column names are case-insensitive.

But how do we know what columns we have in a table? First, 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 the attributes of all columns in the table, such as their 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 = pyfits.open('original.fits', mode='update')
... # making changes in data and/or header
>>> f.flush() # changes are written back to original.fits
```

1.2.2 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 task is very easy in PyFITS 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 = pyfits.PrimaryHDU(n)
```

We then create a HDUList to contain the newly created primary HDU, and write to a new file:

```
>>> hdulist = pyfits.HDUList([hdu])
>>> hdulist.writeto('new.fits')
```

That's it! In fact, PyFITS even provides a short cut for the last two lines to accomplish the same behavior:

```
>>> hdu.writeto('new.fits')
```

This will write a single HDU to a FITS file without having to manually encapsulate it in an `HDUList` object first.

Creating a New Table File

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:

```
>>> import pyfits
>>> import numpy as np
>>> a1 = np.array(['NGC1001', 'NGC1002', 'NGC1003'])
>>> a2 = np.array([11.1, 12.3, 15.2])
>>> col1 = pyfits.Column(name='target', format='20A', array=a1)
>>> col2 = pyfits.Column(name='V_mag', format='E', array=a2)
```

Next, create a `ColDefs` (column-definitions) object for all columns:

```
>>> cols = pyfits.ColDefs([col1, col2])
```

Now, create a new binary table HDU object by using the PyFITS function `new_table()`:

```
>>> tbhdu = pyfits.new_table(cols)
```

This function returns (in this case) a `BinTableHDU`.

Of course, you can do this more concisely without creating intermediate variables for the individual columns and without manually creating a `ColDefs` object:

```
>>> tbhdu = pyfits.new_table(pyfits.ColDefs([pyfits.Column(name='target',
...                                                        format='20A',
...                                                        array=a1),
...                                           pyfits.Column(name='V_mag',
...                                                        format='E',
...                                                        array=a2)]
...                                          ))
```

Now you may write this new table HDU directly to a FITS file like so:

```
>>> tbhdu.writeto('table.fits')
```

This shortcut will automatically create a minimal primary HDU with no data and prepend it to the table HDU to create a valid FITS file. If you require additional data or header keywords in the primary HDU you may still create a `PrimaryHDU` object and build up the FITS file manually using an `HDUList`.

For example, first create a new `Header` object to encapsulate any keywords you want to include in the primary HDU, then as before create a `PrimaryHDU`:

```
>>> prihdr = pyfits.Header()
>>> prihdr['OBSERVER'] = 'Edwin Hubble'
>>> prihdr['COMMENT'] = "Here's some commentary about this FITS file."
>>> prihdu = pyfits.PrimaryHDU(header=prihdr)
```

When we create a new primary HDU with a custom header as in the above example, this will automatically include any additional header keywords that are *required* by the FITS format (keywords such as `SIMPLE` and `NAXIS` for example). In general, PyFITS users should not have to manually manage such keywords, and should only create and modify observation-specific informational keywords.

We then create a HDUList containing both the primary HDU and the newly created table extension, and write to a new file:

```
>>> thdulist = pyfits.HDUList([prihdu, tbhdu])
>>> thdulist.writeto('table.fits')
```

Alternatively, we can append the table to the HDU list we already created in the image file section:

```
>>> hdulist.append(tbhdu)
>>> hdulist.writeto('image_and_table.fits')
```

So far, we have covered the most basic features of PyFITS. In the following chapters we'll show more advanced examples and explain options in each class and method.

1.2.3 Convenience Functions

PyFITS 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.

Warning: These functions are useful for interactive Python sessions and simple analysis scripts, but should not be used for application code, as they are highly inefficient. For example, each call to `getval()` requires re-parsing the entire FITS file. Code that makes repeated use of these functions should instead open the file with `pyfits.open()` and access the data structures directly.

The first of these functions is `getheader()`, to get the header of an HDU. Here are several examples of getting the header. Only the file name is required for this function. The rest of the arguments are optional and flexible to specify which HDU the user wants to access:

```
>>> from pyfits 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
>>> getheader('in.fits', 'sci')  # the first HDU with EXTNAME='SCI'
>>> getheader('in.fits', 'sci', 2)  # HDU with EXTNAME='SCI' and EXTVER=2
>>> getheader('in.fits', ('sci', 2))  # use a tuple to do the same
>>> getheader('in.fits', ext=2)  # the second extension
>>> getheader('in.fits', extname='sci')  # first HDU with EXTNAME='SCI'
>>> getheader('in.fits', extname='sci', extver=2)
```

Ambiguous specifications will raise an exception:

```
>>> getheader('in.fits', ext=('sci', 1), extname='err', extver=2)
...
TypeError: Redundant/conflicting extension arguments(s): {'ext': ('sci',
1), 'args': (), 'extver': 2, 'extname': 'err'}
```

After you get the header, you can access the information in it, such as getting and modifying a keyword value:

```
>>> from pyfits 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 pyfits import getval
>>> flt = getval('in.fits', 'filter', 1)  # get 1st extension's FILTER
>>> val = getval('in.fits', 10, 'sci', 2)  # get the value 2nd SCI
...                                        # extension's 11th keyword
```

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 pyfits 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:

```
>>> pyfits.writeto('out.fits', data, header)
```

The `writeto()` function uses the provided data and an optional header to write to an output FITS file.

```
>>> pyfits.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 pyfits 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:

```
>>> pyfits.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
```

This is one of the most useful convenience functions for getting an overview of what a given file contains without looking at any of the details.

1.3 FITS Headers

In the next three chapters, more detailed information as well as examples will be explained for manipulating FITS headers, image/array data, and table data respectively.

Note: The Header interface was significantly reworked in PyFITS 3.1. This documentation addresses the interface as it works in PyFITS 3.1, though should still be *mostly* the same for earlier PyFITS versions. See the *Header Interface Transition Guide* for a complete detailing on the difference between PyFITS 3.1 and the earlier Header interface. This guide may also be useful to advanced users wishing to better understand how the Header interface is implemented.

1.3.1 Header of an HDU

Every HDU normally has two components: header and data. In PyFITS these two components are accessed through the two attributes of the HDU, `hdu.header` and `hdu.data`.

While an HDU may have empty data, i.e. the `.data` attribute is `None`, any HDU will always have a header. When an HDU is created with a constructor, e.g. `hdu = PrimaryHDU(data, header)`, the user may supply the header value from an existing HDU's header and the data value from a numpy array. If the defaults (None) are used, the new HDU will have the minimal required keywords for an HDU of that type:

```
>>> hdu = pyfits.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. PyFITS will strip any keywords that describe the data structure leaving only your informational keywords. Later it will add back in the required structural keywords for compatibility with the new HDU and any data added to it. So, a user can use a table HDU's header to construct an image HDU and vice versa. The constructor will also ensure the data type and dimension information in the header agree with the data.

1.3.2 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 = pyfits.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's `dict` type, 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.

Keyword Wildcards

Some operations on header keywords also work on keyword wildcard patterns similar to those used to match files in a command shell–the character `'?'` may be used to match a single unknown character, and `'*'` may be used to match zero or more characters. For example:

```
>>> header = pyfits.Header([('SIMPLE', True), ('NAXIS', 2),
...                         ('NAXIS1', 1000), ('NAXIS2', 2000)])
>>> header
SIMPLE  =                    T
NAXIS   =                    2
NAXIS1  =                 1000
NAXIS2  =                 2000
>>> header['NAXIS*']
NAXIS   =                    2
NAXIS1  =                 1000
NAXIS2  =                 2000
```

The object from the above example is a new `Header` object containing only the cards that match the wildcard pattern. It can be thought of like slicing a list, only the slice is non-linear. To return only the cards with `NAXISn` keywords (that is, "NAXIS" followed by one or more characters):

```
>>> header['NAXIS?*']
NAXIS1  =                 1000
NAXIS2  =                 2000
```

Wildcards also work for assignment and deletion:

```
>>> header['NAXIS?*'] = 3000
>>> header
SIMPLE  =                    T
NAXIS   =                    2
NAXIS1  =                 3000
NAXIS2  =                 3000
>>> del header['NAXIS?*']
>>> header
SIMPLE  =                    T
NAXIS   =                    2
```

1.3.3 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 PyFITS, 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 = pyfits.Card('TEMP', 80.0, 'temperature, floating value')
>>> c2 = pyfits.Card('DETECTOR', 1)  # comment is optional
>>> c3 = pyfits.Card('MIR_REVR', True, 'mirror reversed? Boolean value')
>>> c4 = pyfits.Card('ABC', 2+3j, 'complex value')
>>> c5 = pyfits.Card('OBSERVER', 'Hubble', 'string value')

>>> print c1; print c2; print c3; print c4; print c5  # show the cards
```

```
TEMP = 80.0 / temperature, floating value
DETECTOR= 1 /
MIR_REVR= T / mirror reversed? Boolean value
ABC = (2.0, 3.0) / complex value
OBSERVER= 'Hubble ' / string value
```

Cards have the attributes `.keyword`, `.value`, and `.comment`. Both `.value` and `.comment` can be changed but not the `.keyword` attribute. In other words, once a card is created, it is created for a specific, immutable keyword.

The `Card()` constructor will check if the arguments given are conforming to the FITS standard and has a fixed card image format. If the user wants to create a card with a customized format or even a card which is not conforming to the FITS standard (e.g. for testing purposes), the `Card.fromstring()` class method can be used.

Cards can be verified with `Card.verify()`. The non-standard card `c2` in the example below is flagged by such verification. More about verification in PyFITS will be discussed in a later chapter.

```
>>> c1 = pyfits.Card.fromstring('ABC = 3.456D023')
>>> c2 = pyfits.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.

1.3.4 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. PyFITS 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 = pyfits.Header()
>>> header['abc'] = 'abcdefg' * 20
>>> header
ABC = 'abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcd&'
CONTINUE 'efgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefga&'
CONTINUE 'bcdefg&'
>>> header['abc']
'abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgab
>>> # 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 PyFITS 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 PyFITS 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.

1.3.5 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. PyFITS supports this convention as well.

If a keyword contains more than 8 characters PyFITS 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 = pyfits.Card('abcdefghi', 10)
Keyword name 'abcdefghi' is greater than 8 characters; a HIERARCH card will
be created.
>>> print c
HIERARCH abcdefghi = 10
>>> c = pyfits.Card('hierarch abcdefghi', 10)
>>> print c
HIERARCH abcdefghi = 10
>>> h = pyfits.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 "pyfits/header.py", line 121, in __getitem__
    return self._cards[self._cardindex(key)].value
  File "pyfits/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, PyFITS 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.

1.4 Image Data

In this chapter, we'll discuss the data component in an image HDU.

1.4.1 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 PyFITS, 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 = pyfits.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.

1.4.2 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 PyFITS is a numpy array of the same data type 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 = pyfits.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 `pyfits.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 even when the physical values are updated. In other words, it reapplies the scaling to the new physical values upon saving.

Writing Scaled Image Data

With the extra processing and memory requirement, we discourage use of scaled data as much as possible. However, PyFITS does provide ways to write scaled data with the `scale` 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 = pyfits.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')
```

1.4.3 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 larger than 2 to 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 = pyfits.open('file1.fits')
>>> f2 = pyfits.open('file2.fits')
>>> f3 = pyfits.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 main memory.

Sections cannot currently be assigned to. Any modifications made to a data section are not saved back to the original file.

1.5 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 take up more storage space as well as more processing time since the ASCII text needs to be parsed into numerical values.

1.5.1 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 = pyfits.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 PyFITS, 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.

1.5.2 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:

```
>>> import pyfits
>>> t = pyfits.open('table.fits')
>>> tbdata = t[1].data
>>> mask = tbdata.field('magnitude') > 5
>>> newtbdata = tbdata[mask]
>>> hdu = pyfits.BinTableHDU(newtbdata)
>>> hdu.writeto('newtable.fits')
```

Merging Tables

Merging different tables is straightforward in PyFITS. Simply merge the column definitions of the input tables:

```
>>> t1 = pyfits.open('table1.fits')
>>> t2 = pyfits.open('table2.fits')
# the column attribute is the column definitions
>>> t = t1[1].columns + t2[1].columns
>>> hdu = pyfits.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.

A simpler version of this example can be used to append a new column to a table. Updating an existing table with a new column is generally more difficult than it's worth, but one can "append" a column to a table by creating a new table with columns from the existing table plus the new column(s):

```
>>> orig_table = pyfits.open('table.fits')[1].data
>>> orig_cols = orig_table.columns
>>> new_cols = pyfits.ColDefs([
...     pyfits.Column(name='NEWCOL1', format='D',
...                   array=np.zeros(len(orig_table))),
...     pyfits.Column(name='NEWCOL2', format='D',
...                   array=np.zeros(len(orig_table)))])
>>> hdu = pyfits.new_table(orig_cols + new_cols)
>>> hdu.writeto('newtable.fits')
```

Now `newtable.fits` contains a new table with the original table, plus the two new columns filled with zeros.

Appending Tables

Appending one table after another is slightly trickier, since the two tables may have different field attributes. 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 = pyfits.open('table1.fits')
>>> t2 = pyfits.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 = pyfits.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')
```

1.5.3 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.

1.5.4 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 pyfits 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 = pyfits.ColDefs([c1, c2, c3, c4, c5])
>>> tbhdu = pyfits.new_table(coldefs)
```

or directly use the `new_table()` function:

```
>>> tbhdu = pyfits.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 = pyfits.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 PyFITS table creation will be simplified and this process won't be necessary.

1.6 Verification

PyFITS has built in a flexible scheme to verify FITS data being conforming to the FITS standard. The basic verification philosophy in PyFITS is to be tolerant in input and strict in output.

When PyFITS 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.

1.6.1 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. PyFITS's verification scheme is developed on these hierarchical levels. Here are the 3 PyFITS 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 PyFITS 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 PyFITS 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.

1.6.2 Verification Options

There are 5 options for all verify(option) calls in PyFITS. 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 PyFITS will not take such action to hurt the integrity of the data.

Not all fixes may be the "correct" fix, but at least PyFITS 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.

1.6.3 Verifications at Different Data Object Levels

We'll examine what PyFITS'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 '
>>> hdu.header[5]  # can still access the values before the fix
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 = pyfits.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 = pyfits.Card.fromstring("P.I. = 'Hubble'")
>>> h.header.append(c)  # then append it to the header
>>> # Now if we try to write to a FITS file, the default output
>>> # verification will not take it.
>>> h.writeto('pi.fits')
Output verification result:
HDU 0:
  Card 4:
    Unfixable error: Illegal keyword name 'P.I.'
......
  raise VerifyError
VerifyError
```

```
# Must set the output_verify argument to 'ignore', to force writing a
# non-standard FITS file
>>> h.writeto('pi.fits', output_verify='ignore')
# Now reading a non-standard FITS file
# pyfits is magnanimous in reading non-standard FITS file
>>> hdus = pyfits.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.'
```

1.6.4 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 PyFITS 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 = pyfits.open('pix.fits', checksum=True)

>>> # Open the file in.fits where checksum verification fails for the
>>> # primary HDU
```

```
>>> hdul = pyfits.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.
>>> pyfits.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.
>>> pyfits.append('existingfile.fits', data, checksum=True)
```

1.7 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.

1.7.1 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 PyFITS, 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, PyFITS will automatically detect what kind of table it is.

```
>>> hdus = pyfits.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
>>> import pyfits
>>> a1 = np.array(['abcd', 'def'])
>>> r1 = np.array([11., 12.])
>>> c1 = pyfits.Column(name='abc', format='A3', array=a1)
>>> c2 = pyfits.Column(name='def', format='E', array=r1, bscale=2.3,
...                    bzero=0.6)
>>> c3 = pyfits.Column(name='t1', format='I', array=[91, 92, 93])
# Create the table
>>> x = pyfits.ColDefs([c1, c2, c3], tbtype='TableHDU')
>>> hdu = pyfits.new_table(x, tbtype='TableHDU')
# Or, simply,
>>> hdu = pyfits.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')])
```

1.7.2 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. PyFITS 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 PyFITS), 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 = pyfits.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.

```
>>> import pyfits
>>> import numpy as np
>>> c1 = pyfits.Column(name='var', format='PJ()',
...                    array=np.array([[45., 56]
                                       [11, 12, 13]],
...                                   dtype=np.object))
>>> c2 = pyfits.Column(name='xyz', format='2I', array=[[11, 3], [12, 4]])
# the rest is the same as a regular table.
# Create the table HDU
>>> tbhdu = pyfits.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 = pyfits.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 '
```

1.7.3 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 = pyfits.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 = pyfits.GroupData(imdata, parnames=['abc', 'xyz'],
...                      pardata=[pdata1, pdata2], bitpix=-32)
# Now, create the GroupsHDU and write to a FITS file.
>>> hdu = pyfits.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))
]
```

1.7.4 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 PyFITS, using the optional "pyfits.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 pyfitsComp 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 pyfitsComp module is not required in order for PyFITS to work.

Header and Summary

In PyFITS, 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 = pyfits.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 = pyfits.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:

```
>>> pyfits.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 = pyfits.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 = pyfits.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 = pyfits.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.

1.8 Executable Scripts

PyFITS installs a couple of useful utility programs on your system that are built with PyFITS.

1.8.1 fitscheck

`fitscheck` is a command line script based on pyfits 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 pyfits 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
   ```

Command u'fitscheck –help' failed: [Errno 2] No such file or directory

1.8.2 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:

Command u'fitsdiff –help' failed: [Errno 2] No such file or directory

1.9 Miscellaneous Features

In this chapter, we'll describe some of the miscellaneous features of PyFITS.

1.9.1 Warning Messages

PyFITS 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
import pyfits

# ...

warnings.resetwarnings()
warnings.filterwarnings('ignore', category=UserWarning, append=True)
pyfits.writeto(file, im, clobber=True)
warnings.resetwarnings()
warnings.filterwarnings('always', category=UserWarning, append=True)

# ...
```

PyFITS 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.

1.9.2 Differs

The `pyfits.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.

1.10 Reference Manual

Examples

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

Figure 1.1: Red color information

```
#!/usr/bin/env python
import pyfits
import numpy
import Image

#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
```

Figure 1.2: Green color information

Figure 1.3: Blue color information

```
# scalings; don't use for science
red = pyfits.PrimaryHDU(data=npr)
red.header['LATOBS'] = "32:11:56" # add spurious header info
red.header['LONGOBS'] = "110:56"
red.writeto('red.fits')

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

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

CHAPTER 2

API Documentation

2.1 File Handling and Convenience Functions

2.1.1 open()

pyfits.**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.

2.1.2 writeto()

pyfits.**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 in a writeable binary mode such as 'wb' or 'ab+'.

data : array, record array, or groups data object

data to write to the new file

header : Header object, optional

the header associated with data. If None, a header of the appropriate type is created for the supplied data. This argument is optional.

output_verify : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". 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.

2.1.3 info()

`pyfits.`**`info`**(*filename*, *output=None*, ***kwargs*)

Print the summary information on a FITS file.

This includes the name, type, length of header, data shape and type for each extension.

Parameters

filename : file path, file object, or file like object

FITS file to obtain info from. If opened, mode must be one of the following: rb, rb+, or ab+ (i.e. the file must be readable).

output : file, bool, optional

A file-like object to write the output to. If `False`, does not output to a file and instead returns a list of tuples representing the HDU info. Writes to `sys.stdout` by default.

kwargs :

Any additional keyword arguments to be passed to `pyfits.open`. *Note:* This function sets `ignore_missing_end=True` by default.

2.1.4 append()

`pyfits.`**`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 `pyfits.open`.

2.1.5 update()

`pyfits.`**`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 5th extension
```

kwargs :

Any additional keyword arguments to be passed to `pyfits.open`.

2.1.6 getdata()

`pyfits.`**`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 returned wrapped in the given ndarray subclass by calling:

```
data.view(view)
```

kwargs :

Any additional keyword arguments to be passed to pyfits.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.

2.1.7 getheader()

pyfits.**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 `pyfits.open`.

Returns

header : `Header` object

2.1.8 getval()

`pyfits.`**`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 `pyfits.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

2.1.9 setval()

`pyfits.`**`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 `pyfits.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.

2.1.10 `delval()`

`pyfits.`**`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

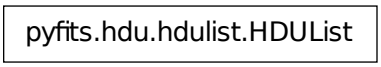
ext, extname, extver :

The rest of the arguments are for extension specification. See `getdata` for explanations/examples.

kwargs :

Any additional keyword arguments to be passed to `pyfits.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.

2.2 HDU Lists



2.2.1 HDUList

class pyfits.**HDUList**(*hdus=[ ]*, *file=None*)

Bases: list, pyfits.verify._Verify

HDU list class. This is the top-level FITS object. When a FITS file is opened, a HDUList object is returned.

Construct a HDUList object.

Parameters

hdus : sequence of HDU objects or single HDU, optional

The HDU object(s) to comprise the HDUList. Should be instances of HDU classes like ImageHDU or BinTableHDU.

file : file object, optional

The opened physical file associated with the HDUList.

append(*hdu*)

Append a new HDU to the HDUList.

Parameters

hdu : HDU object

HDU to add to the HDUList.

close(*output_verify='exception'*, *verbose=False*, *closed=True*)

Close the associated FITS file and memmap object, if any.

Parameters

output_verify : str

Output verification option. Must be one of "fix", "silentfix", "ignore", "warn", or "exception". 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 `pyfits.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 `pyfits.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, bool, optional

A file-like object to write the output to. If `False`, does not output to a file and instead returns a list of tuples representing the HDU info. Writes to `sys.stdout` by default.

insert(*index*, *hdu*)

Insert an HDU into the `HDUList` at the given `index`.

Parameters

index : int

Index before which to insert the new HDU.

hdu : HDU object

The HDU object to insert

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.

2.3 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*.

2.3.1 `PrimaryHDU`

class `pyfits.`**`PrimaryHDU`**(*data=None*, *header=None*, *do_not_scale_image_data=False*, *uint=False*, *scale_back=None*)

Bases: `pyfits.hdu.image._ImageBaseHDU`

FITS primary HDU class.

Construct a primary HDU.

Parameters

data : array or DELAYED, optional

The data in the HDU.

header : Header instance, optional

The header to be used (as a template). If `header` is `None`, a minimal header will be provided.

do_not_scale_image_data : bool, optional

If `True`, image data is not scaled using BSCALE/BZERO values when read.

uint : bool, optional

Interpret signed integer data where `BZERO` is the central value and `BSCALE == 1` as unsigned integer data. For example, `int16` data with `BZERO = 32768` and `BSCALE = 1` would be treated as `uint16` data.

scale_back : bool, optional

If `True`, when saving changes to a file that contained scaled image data, restore the data to the original type and reapply the original BSCALE/BZERO values. This could lead to loss of accuracy if scaling back to integer values after performing floating point operations on the data.

`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.

fileinfo()

Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the `HDUList`.

Returns

dict or None :

The dictionary details information about the locations of this HDU within an associated file. Returns `None` when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
file-mode	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.

classmethod **`readfrom`**(*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with `pyfits.open` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto`.

Parameters

fileobj : file object or file-like object

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : bool

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : bool

Do not issue an exception when opening a file that is missing an `END` card in the last header.

req_cards(*keyword*, *pos*, *test*, *fix_value*, *option*, *errlist*)
Check the existence, location, and value of a required `Card`.

Parameters

keyword : str

The keyword to validate

pos : int, callable

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument–the actual position of the keyword–and return `True` or `False`. This can be used for custom evaluation. For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : callable

This should be a callable (generally a function) that is passed the value of the given keyword and returns `True` or `False`. This can be used to validate the value associated with the given keyword.

fix_value : str, int, float, complex, bool, None

A valid value for a FITS keyword to to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If `None`, there is no replacement value and the keyword is unfixable.

option : str

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. See *Verification options* for more info.

errlist : list

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to `req_cards`.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

run_option(*option='warn'*, *err_text=''*, *fix_text='Fixed.'*, *fix=None*, *fixable=True*)
Execute the verification with selected option.

scale(*type=None*, *option='old'*, *bscale=1*, *bzero=0*)
Scale image data by using `BSCALE`/`BZERO`.

Call to this method will scale `data` and update the keywords of `BSCALE` and `BZERO` in the HDU's header. This method should only be used right before writing to the output file, as the data will be scaled and is therefore not very usable after the call.

Parameters

type : str, optional

destination data type, use a string representing a numpy dtype name, (e.g. `'uint8'`, `'int16'`, `'float32'` etc.). If is `None`, use the current data type.

option : str

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

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 are 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 are 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.

data
Image/array data as a `ndarray`.

Please remember that the order of axes on an Numpy array are opposite of the order specified in the FITS file. For example for a 2D image the "rows" or y-axis are the first dimension, and the "columns" or x-axis are the second dimension.

If the data is scaled using the BZERO and BSCALE parameters, this attribute returns the data scaled to its physical values unless the file was opened with `do_not_scale_image_data=True`.

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.

2.3.2 GroupsHDU

class `pyfits.`**GroupsHDU**(*data=None*, *header=None*, *name=None*)
Bases: `pyfits.hdu.image.PrimaryHDU`, `pyfits.hdu.table._TableLikeHDU`

FITS Random Groups HDU class.

See the *Random Access Groups* section in the PyFITS documentation for more details on working with this type of HDU.

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.

fileinfo()

Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the `HDUList`.

Returns

dict or None :

The dictionary details information about the locations of this HDU within an associated file. Returns `None` when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
file-mode	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.

classmethod readfrom(*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with `pyfits.open` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto`.

Parameters

fileobj : file object or file-like object

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : bool

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : bool

Do not issue an exception when opening a file that is missing an `END` card in the last header.

req_cards(*keyword*, *pos*, *test*, *fix_value*, *option*, *errlist*)

Check the existence, location, and value of a required `Card`.

Parameters

keyword : str

The keyword to validate

pos : int, callable

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument–the actual position of the keyword–and return `True` or `False`. This can be used for custom evaluation.

For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : callable

This should be a callable (generally a function) that is passed the value of the given keyword and returns `True` or `False`. This can be used to validate the value associated with the given keyword.

fix_value : str, int, float, complex, bool, None

A valid value for a FITS keyword to to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If `None`, there is no replacement value and the keyword is unfixable.

option : str

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. See *Verification options* for more info.

errlist : list

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to `req_cards`.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

run_option(*option='warn'*, *err_text=''*, *fix_text='Fixed.'*, *fix=None*, *fixable=True*)
Execute the verification with selected option.

scale(*type=None*, *option='old'*, *bscale=1*, *bzero=0*)
Scale image data by using `BSCALE`/`BZERO`.

Call to this method will scale `data` and update the keywords of `BSCALE` and `BZERO` in the HDU's header. This method should only be used right before writing to the output file, as the data will be scaled and is therefore not very usable after the call.

Parameters

type : str, optional

destination data type, use a string representing a numpy dtype name, (e.g. `'uint8'`, `'int16'`, `'float32'` etc.). If is `None`, use the current data type.

option : str

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

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 are 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 are 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.

data
The data of a random group FITS file will be like a binary table's data.

parnames
The names of the group parameters as described by the header.

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.

2.3.3 GroupData

class `pyfits.`**GroupData**
Bases: `pyfits.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.

2.3.4 Group

class `pyfits.`**GroupData**
Bases: `pyfits.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.

2.3.5 StreamingHDU

class `pyfits.`**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 = pyfits.Header()

for all the cards you need in the header:
    header[key] = (value, comment)

shdu = pyfits.StreamingHDU('filename.fits', header)

for each piece of data:
    shdu.write(data)
```

```
shdu.close()
```

Construct a `StreamingHDU` object given a file name and a header.

Parameters

name : file path, file object, or file like object

The file to which the header and data will be streamed. If opened, the file object must be opened in a writeable binary mode such as 'wb' or 'ab+'.

header : `Header` instance

The header object associated with the data to be written to the file.

Notes

The file will be opened and the header appended to the end of the file. If the file does not already exist, it will be created, and if the header represents a Primary header, it will be written to the beginning of the file. If the file does not exist and the provided header is not a Primary header, a default Primary HDU will be inserted at the beginning of the file and the provided header will be added as the first extension. If the file does already exist, but the provided header represents a Primary header, the header will be modified to an image extension header and appended to the end of the file.

close()
Close the physical FITS file.

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.

size
Return the size (in bytes) of the data portion of the HDU.

2.4 Headers

2.4.1 Header

class pyfits.**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 PyFITS documentation for more details on working with headers.

Construct a `Header` from an iterable and/or text file.

Parameters

cards : A list of `Card` objects, optional

The cards to initialize the header with.

txtfile : file path, file object or file-like object, optional

Input ASCII header parameters file **(Deprecated)** Use the Header.fromfile classmethod instead.

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.

ascardlist(**args*, ***kwargs*)
Deprecated since version 3.0: Use the `ascard` attribute instead.

Returns a `CardList` object.

clear()
Remove all cards from the header.

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(*key*, *card*, *useblanks=True*, *after=False*)

Inserts a new keyword+value card into the Header at a given location, similar to `list.insert`.

Parameters

key : int, str, or tuple

The index into the the list of header keywords before which the new keyword should be inserted, or the name of a keyword before which the new keyword should be inserted. Can also accept a (keyword, index) tuple for inserting around duplicate keywords.

card : str, tuple

A keyword or a (keyword, value, [comment]) tuple; see `Header.append`

useblanks : bool, optional

If there are blank cards at the end of the Header, replace the first blank card so that the total number of cards in the Header does not increase. Otherwise preserve the number of blank cards.

after : bool, optional

If set to `True`, insert *after* the specified index or keyword, rather than before it. Defaults to `False`.

items()

Like `dict.items()`.

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 `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()

Similar to `dict.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*)
Similar to `dict.setitem()`.

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.

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`).

cards
The underlying physical cards that make up this Header; it can be looked at, but it should not be modified directly.

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'
```

2.5 Cards

2.5.1 Card

class pyfits.**Card**(*keyword=None*, *value=None*, *comment=None*, ***kwargs*)
Bases: `pyfits.verify._Verify`

ascardimage(**args*, ***kwargs*)
Deprecated since version 3.1: Use the `image` attribute instead.

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).

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

run_option(*option='warn'*, *err_text=''*, *fix_text='Fixed.'*, *fix=None*, *fixable=True*)
Execute the verification with selected option.

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.

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.

image
The card "image", that is, the 80 byte character string that represents this card in an actual FITS header.

key
Deprecated since version 3.1: Use the `keyword` attribute instead.

keyword
Returns the keyword name parsed from the card image.

length = 80
The length of a Card image; should always be 80 for valid FITS files.

rawkeyword
On record-valued keyword cards this is the name of the standard <= 8 character FITS keyword that this RVKC is stored in. Otherwise it is the card's normal keyword.

rawvalue
On record-valued keyword cards this is the raw string value in the `<field-specifier>: <value>` format stored in the card in order to represent a RVKC. Otherwise it is the card's normal value.

value
The value associated with the keyword stored in this card.

2.5.2 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 `pyfits.`**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.

Construct the `CardList` object from a list of `Card` objects.

`CardList` is now merely a thin wrapper around `Header` to provide backwards compatibility for the old API. This should not be used for any new code.

Parameters
cards :

A list of `Card` objects.

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`.

pyfits.**create_card**(**args*, ***kwargs*)
Deprecated since version 3.1: Use `Card.__init__` instead.

pyfits.**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).

pyfits.**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

2.6 Tables

2.6.1 BinTableHDU

class pyfits.**BinTableHDU**(*data=None*, *header=None*, *name=None*)
Bases: `pyfits.hdu.table._TableBaseHDU`

Binary table HDU class.

Parameters
header : Header instance

header to be used

data : array

data to be used

name : str

name to be populated in `EXTNAME` keyword

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.

fileinfo()
Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the HDUList.

Returns
dict or None :

The dictionary details information about the locations of this HDU within an associated file. Returns None when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
filemode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
hdrLoc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

classmethod fromstring(*data*, *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 `pyfits.open` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto`.

Parameters

fileobj : file object or file-like object

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : bool

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : bool

Do not issue an exception when opening a file that is missing an `END` card in the last header.

`req_cards`(*keyword*, *pos*, *test*, *fix_value*, *option*, *errlist*)

Check the existence, location, and value of a required `Card`.

Parameters

keyword : str

The keyword to validate

pos : int, callable

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument–the actual position of the keyword–and return `True` or `False`. This can be used for custom evaluation. For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : callable

This should be a callable (generally a function) that is passed the value of the given keyword and returns `True` or `False`. This can be used to validate the value associated with the given keyword.

fix_value : str, int, float, complex, bool, None

A valid value for a FITS keyword to to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If `None`, there is no replacement value and the keyword is unfixable.

option : str

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. See *Verification options* for more info.

errlist : list

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to `req_cards`.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

run_option(*option='warn'*, *err_text=''*, *fix_text='Fixed.'*, *fix=None*, *fixable=True*)
Execute the verification with selected option.

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 are 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 are 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).

columns

The ColDefs objects describing the columns in this table.

size

Size (in bytes) of the data portion of the HDU.

2.6.2 TableHDU

class pyfits.**TableHDU**(*data=None*, *header=None*, *name=None*)

Bases: pyfits.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.

fileinfo()

Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the `HDUList`.

Returns

dict or None :

The dictionary details information about the locations of this HDU within an associated file. Returns `None` when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
file-mode	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 `pyfits.open` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto`.

Parameters

fileobj : file object or file-like object

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : bool

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : bool

Do not issue an exception when opening a file that is missing an `END` card in the last header.

req_cards (*keyword*, *pos*, *test*, *fix_value*, *option*, *errlist*)

Check the existence, location, and value of a required `Card`.

Parameters

keyword : str

The keyword to validate

pos : int, callable

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument–the actual position of the keyword–and return `True` or `False`. This can be used for custom evaluation. For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : callable

This should be a callable (generally a function) that is passed the value of the given keyword and returns `True` or `False`. This can be used to validate the value associated with the given keyword.

fix_value : str, int, float, complex, bool, None

A valid value for a FITS keyword to to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If `None`, there is no replacement value and the keyword is unfixable.

option : str

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. See *Verification options* for more info.

errlist : list

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to `req_cards`.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

run_option(*option='warn'*, *err_text=''*, *fix_text='Fixed.'*, *fix=None*, *fixable=True*)
Execute the verification with selected option.

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 are 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 are 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).

columns
The ColDefs objects describing the columns in this table.

size
Size (in bytes) of the data portion of the HDU.

2.6.3 Column

class pyfits.**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.

Construct a Column by specifying attributes. All attributes except format can be optional.

Parameters
name : str, optional

column name, corresponding to TTYPE keyword

format : str, optional

column format, corresponding to TFORM keyword

unit : str, optional

column unit, corresponding to TUNIT keyword

null : str, optional

null value, corresponding to TNULL keyword

bscale : int-like, optional

bscale value, corresponding to TSCAL keyword

bzero : int-like, optional

bzero value, corresponding to TZERO keyword

disp : str, optional

display format, corresponding to `TDISP` keyword

start : int, optional

column starting position (ASCII table only), corresponding to `TBCOL` keyword

dim : str, optional

column dimension corresponding to `TDIM` keyword

copy()
Return a copy of this `Column`.

2.6.4 ColDefs

class pyfits.**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.

Parameters
input : :

An existing table HDU, an existing ColDefs, or recarray

****(Deprecated)** tbtype** : str, optional

which table HDU, `"BinTableHDU"` (default) or `"TableHDU"` (text table). Now ColDefs for a normal (binary) table by default, but converted automatically to ASCII table ColDefs in the appropriate contexts (namely, when creating an ASCII table).

add_col(*column*)
Append one `Column` to the column definition.

> **Warning:** *New in pyfits 2.3*: This function appends the new column to the `ColDefs` object in place. Prior to pyfits 2.3, this function returned a new `ColDefs` with the new column at the end.

change_attrib(*col_name*, *attrib*, *new_value*)
Change an attribute (in the `KEYWORD_ATTRIBUTES` list) of a `Column`.

Parameters
col_name : str or int

The column name or index to change

attrib : str

The attribute name

value : object

The new value for the attribute

change_name(*col_name*, *new_name*)
Change a `Column`'s name.

Parameters
col_name : str

The current name of the column

new_name : str

The new name of the column

change_unit(*col_name*, *new_unit*)
Change a Column's unit.

Parameters
col_name : str or int

The column name or index

new_unit : str

The new unit for the column

del_col(*col_name*)
Delete (the definition of) one Column.

col_name
[str or int] The column's name or index

info(*attrib='all'*, *output=None*)
Get attribute(s) information of the column definition.

Parameters
attrib : str

Can be one or more of the attributes listed in pyfits.column.KEYWORD_ATTRIBUTES. The default is "all" which will print out all attributes. It forgives plurals and blanks. If there are two or more attribute names, they must be separated by comma(s).

output : file, optional

File-like object to output to. Outputs to stdout by default. If False, returns the attributes as a dict instead.

Notes

This function doesn't return anything by default; it just prints to stdout.

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.

2.6.5 FITS_rec

class pyfits.**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.

field(*key*)
A view of a Column's data as an array.

columns
A user-visible accessor for the coldefs.

See https://trac.assembla.com/pyfits/ticket/44

2.6.6 FITS_record

class pyfits.**FITS_record**(*input*, *row=0*, *start=None*, *end=None*, *step=None*, *base=None*, ***kwargs*)
Bases: object

FITS record class.

FITS_record is used to access records of the FITS_rec object. This will allow us to deal with scaled columns. It also handles conversion/scaling of columns in ASCII tables. The FITS_record class expects a FITS_rec object as input.

Parameters
input : array

The array to wrap.

row : int, optional

The starting logical row of the array.

start : int, optional

The starting column in the row associated with this object. Used for subsetting the columns of the FITS_rec object.

end : int, optional

The ending column in the row associated with this object. Used for subsetting the columns of the FITS_rec object.

field(*field*)
Get the field data of the record.

setfield(*field*, *value*)
Set the field data of the record.

2.6.7 Table Functions

new_table()

pyfits.**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 a ColDefs

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()

pyfits.**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 multidimensional array (`TDIMn`). The sixth field provides the value that signifies an undefined value (`TNULLn`). The seventh field provides the scale factor (`TSCALn`). The eighth field provides the offset value (`TZEROn`). A field value of `""` is used to represent the case where no value is provided.

- **hfile:** Each line of the header parameters file provides the definition of a single HDU header card as represented by the card image.

tableload()

pyfits.**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 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.

2.7 Images

2.7.1 `ImageHDU`

class `pyfits.`**`ImageHDU`**(*data=None*, *header=None*, *name=None*, *do_not_scale_image_data=False*, *uint=False*, *scale_back=None*)

Bases: `pyfits.hdu.image._ImageBaseHDU`, `pyfits.hdu.base.ExtensionHDU`

FITS image extension HDU class.

Construct an image HDU.

Parameters

data : array

The data in the HDU.

header : Header instance

The header to be used (as a template). If `header` is `None`, a minimal header will be provided.

name : str, optional

The name of the HDU, will be the value of the keyword `EXTNAME`.

do_not_scale_image_data : bool, optional

If `True`, image data is not scaled using BSCALE/BZERO values when read.

uint : bool, optional

Interpret signed integer data where `BZERO` is the central value and `BSCALE == 1` as unsigned integer data. For example, `int16` data with `BZERO = 32768` and `BSCALE = 1` would be treated as `uint16` data.

scale_back : bool, optional

If `True`, when saving changes to a file that contained scaled image data, restore the data to the original type and reapply the original BSCALE/BZERO values. This could lead to loss of accuracy if scaling back to integer values after performing floating point operations on the data.

`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.

fileinfo()

Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the `HDUList`.

Returns

dict or None :

The dictionary details information about the locations of this HDU within an associated file. Returns `None` when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
file-mode	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.

classmethod **readfrom**(*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)

Read the HDU from a file. Normally an HDU should be opened with `pyfits.open` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto`.

Parameters

fileobj : file object or file-like object

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : bool

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : bool

Do not issue an exception when opening a file that is missing an `END` card in the last header.

req_cards (*keyword*, *pos*, *test*, *fix_value*, *option*, *errlist*)
Check the existence, location, and value of a required `Card`.

Parameters

keyword : str

The keyword to validate

pos : int, callable

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument–the actual position of the keyword–and return `True` or `False`. This can be used for custom evaluation. For example if `pos=lambda idx:  idx > 10` this will check that the keyword's index is greater than 10.

test : callable

This should be a callable (generally a function) that is passed the value of the given keyword and returns `True` or `False`. This can be used to validate the value associated with the given keyword.

fix_value : str, int, float, complex, bool, None

A valid value for a FITS keyword to to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If `None`, there is no replacement value and the keyword is unfixable.

option : str

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. See *Verification options* for more info.

errlist : list

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to `req_cards`.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

run_option (*option='warn'*, *err_text=''*, *fix_text='Fixed.'*, *fix=None*, *fixable=True*)
Execute the verification with selected option.

scale (*type=None*, *option='old'*, *bscale=1*, *bzero=0*)
Scale image data by using `BSCALE`/`BZERO`.

Call to this method will scale `data` and update the keywords of `BSCALE` and `BZERO` in the HDU's header. This method should only be used right before writing to the output file, as the data will be scaled and is therefore not very usable after the call.

Parameters

type : str, optional

destination data type, use a string representing a numpy dtype name, (e.g. `'uint8'`, `'int16'`, `'float32'` etc.). If is `None`, use the current data type.

option : str

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

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 are 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 are 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).

data
Image/array data as a `ndarray`.

Please remember that the order of axes on an Numpy array are opposite of the order specified in the FITS file. For example for a 2D image the "rows" or y-axis are the first dimension, and the "columns" or x-axis are the second dimension.

If the data is scaled using the BZERO and BSCALE parameters, this attribute returns the data scaled to its physical values unless the file was opened with `do_not_scale_image_data=True`.

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.

2.7.2 CompImageHDU

class `pyfits.`**`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: `pyfits.hdu.table.BinTableHDU`

Compressed Image HDU class.

Parameters

data : array, optional

Uncompressed image data

header : Header instance, optional

Header to be associated with the image; when reading the HDU from a file (data=DELAYED), the header read from the file

name : str, optional

The `EXTNAME` value; if this value is `None`, then the name from the input image header will be used; if there is no name in the input image header then the default name `COMPRESSED_IMAGE` is used.

compressionType : str, optional

Compression algorithm: one of `'RICE_1'`, `'RICE_ONE'`, `'PLIO_1'`, `'GZIP_1'`, `'GZIP_2'`, `'HCOMPRESS_1'`

tileSize : int, optional

Compression tile sizes. Default treats each row of image as a tile.

hcompScale : float, optional

HCOMPRESS scale parameter

hcompSmooth : float, optional

HCOMPRESS smooth parameter

quantizeLevel : float, optional

Floating point quantization level; see note below

Notes

The pyfits module supports 2 methods of image compression.

1. The entire FITS file may be externally compressed with the gzip or pkzip utility programs, producing a `*.gz` or `*.zip` file, respectively. When reading compressed files of this type, pyfits first uncompresses the entire file into a temporary file before performing the requested read operations. The pyfits module does not support writing to these types of compressed files. This type of compression is supported in the `_File` class, not in the `CompImageHDU` class. The file compression type is recognized by the `.gz` or `.zip` file name extension.

2. The `CompImageHDU` class supports the FITS tiled image compression convention in which the image is subdivided into a grid of rectangular tiles, and each tile of pixels is individually compressed. The details of this FITS compression convention are described at the FITS Support Office web site. Basically, the compressed image tiles are stored in rows of a variable length arrray column in a FITS binary table. The pyfits module recognizes that this binary table extension contains an image and treats it as if it were an image extension. Under this tile-compression format, FITS header keywords remain uncompressed. At this time, pyfits does not support the ability to extract and uncompress sections of the image without having to uncompress the entire image.

The pyfits module supports 3 general-purpose compression algorithms plus one other special-purpose compression technique that is designed for data masks with positive integer pixel values. The 3 general purpose algorithms are GZIP, Rice, and HCOMPRESS, and the special-purpose technique is the IRAF pixel list compression technique (PLIO). The `compressionType` parameter defines the compression algorithm to be used.

The FITS image can be subdivided into any desired rectangular grid of compression tiles. With the GZIP, Rice, and PLIO algorithms, the default is to take each row of the image as a tile. The HCOMPRESS algorithm is inherently 2-dimensional in nature, so the default in this case is to take 16 rows of the image per tile. In most cases, it makes little difference what tiling pattern is used, so the default tiles are usually adequate. In the case of very small images, it could be more efficient to compress the whole image as a single tile. Note that the image dimensions are not required to be an integer multiple of the tile dimensions; if not, then the tiles at the edges of the image will be smaller than the other tiles. The `tileSize` parameter may be provided as a list of tile sizes, one for each dimension in the image. For example a `tileSize` value of `[100,100]` would divide a 300 X 300 image into 9 100 X 100 tiles.

The 4 supported image compression algorithms are all 'loss-less' when applied to integer FITS images; the pixel values are preserved exactly with no loss of information during the compression and uncompression process. In addition, the HCOMPRESS algorithm supports a 'lossy' compression mode that will produce larger amount of image compression. This is achieved by specifying a non-zero value for the `hcompScale` parameter. Since the amount of compression that is achieved depends directly on the RMS noise in the image, it is usually more convenient to specify the `hcompScale` factor relative to the RMS noise. Setting `hcompScale` = 2.5 means use a scale factor that is 2.5 times the calculated RMS noise in the image tile. In some cases it may be desirable to specify the exact scaling to be used, instead of specifying it relative to the calculated noise value. This may be done by specifying the negative of the desired scale value (typically in the range -2 to -100).

Very high compression factors (of 100 or more) can be achieved by using large `hcompScale` values, however, this can produce undesireable 'blocky' artifacts in the compressed image. A variation of the HCOMPRESS algorithm (called HSCOMPRESS) can be used in this case to apply a small amount of smoothing of the image when it is uncompressed to help cover up these artifacts. This smoothing is purely cosmetic and does not cause any significant change to the image pixel values. Setting the `hcompSmooth` parameter to 1 will engage the smoothing algorithm.

Floating point FITS images (which have `BITPIX` = -32 or -64) usually contain too much 'noise' in the least significant bits of the mantissa of the pixel values to be effectively compressed with any lossless algorithm. Consequently, floating point images are first quantized into scaled integer pixel values (and thus throwing away much of the noise) before being compressed with the specified algorithm (either GZIP, RICE, or HCOMPRESS). This technique produces much higher compression factors than simply using the GZIP utility to externally compress the whole FITS file, but it also means that the original floating point value pixel values are not exactly

perserved. When done properly, this integer scaling technique will only discard the insignificant noise while still preserving all the real imformation in the image. The amount of precision that is retained in the pixel values is controlled by the `quantizeLevel` parameter. Larger values will result in compressed images whose pixels more closely match the floating point pixel values, but at the same time the amount of compression that is achieved will be reduced. Users should experiment with different values for this parameter to determine the optimal value that preserves all the useful information in the image, without needlessly preserving all the 'noise' which will hurt the compression efficiency.

The default value for the `quantizeLevel` scale factor is 16, which means that scaled integer pixel values will be quantized such that the difference between adjacent integer values will be 1/16th of the noise level in the image background. An optimized algorithm is used to accurately estimate the noise in the image. As an example, if the RMS noise in the background pixels of an image = 32.0, then the spacing between adjacent scaled integer pixel values will equal 2.0 by default. Note that the RMS noise is independently calculated for each tile of the image, so the resulting integer scaling factor may fluctuate slightly for each tile. In some cases, it may be desireable to specify the exact quantization level to be used, instead of specifying it relative to the calculated noise value. This may be done by specifying the negative of desired quantization level for the value of `quantizeLevel`. In the previous example, one could specify `quantizeLevel'=-2.0 so that the quantized integer levels differ by 2.0. Larger negative values for 'quantizeLevel` means that the levels are more coarsely-spaced, and will produce higher compression factors.

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.

fileinfo()
Returns a dictionary detailing information about the locations of this HDU within any associated file. The values are only valid after a read or write of the associated file with no intervening changes to the HDUList.

Returns

dict or None :

The dictionary details information about the locations of this HDU within an associated file. Returns None when the HDU is not associated with a file.

Dictionary contents:

Key	Value
file	File object associated with the HDU
filemode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
hdrLoc	Starting byte location of header in file
datLoc	Starting byte location of data block in file
datSpan	Data size including padding

classmethod fromstring(*data*, *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 `pyfits.open` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto`.

Parameters

fileobj : file object or file-like object

Input FITS file. The file's seek pointer is assumed to be at the beginning of the HDU.

checksum : bool

If `True`, verifies that both `DATASUM` and `CHECKSUM` card values (when present in the HDU header) match the header and data of all HDU's in the file.

ignore_missing_end : bool

Do not issue an exception when opening a file that is missing an `END` card in the last header.

`req_cards`(*keyword*, *pos*, *test*, *fix_value*, *option*, *errlist*)

Check the existence, location, and value of a required `Card`.

Parameters

keyword : str

The keyword to validate

pos : int, callable

If an `int`, this specifies the exact location this card should have in the header. Remember that Python is zero-indexed, so this means `pos=0` requires the card to be the first card in the header. If given a callable, it should take one argument–the actual position of the keyword–and return `True` or `False`. This can be used for custom evaluation. For example if `pos=lambda idx: idx > 10` this will check that the keyword's index is greater than 10.

test : callable

This should be a callable (generally a function) that is passed the value of the given keyword and returns `True` or `False`. This can be used to validate the value associated with the given keyword.

fix_value : str, int, float, complex, bool, None

A valid value for a FITS keyword to to use if the given `test` fails to replace an invalid value. In other words, this provides a default value to use as a replacement if the keyword's current value is invalid. If `None`, there is no replacement value and the keyword is unfixable.

option : str

Output verification option. Must be one of `"fix"`, `"silentfix"`, `"ignore"`, `"warn"`, or `"exception"`. See *Verification options* for more info.

errlist : list

A list of validation errors already found in the FITS file; this is used primarily for the validation system to collect errors across multiple HDUs and multiple calls to `req_cards`.

Notes

If `pos=None`, the card can be anywhere in the header. If the card does not exist, the new card will have the `fix_value` as its value when created. Also check the card's value by using the `test` argument.

run_option(*option='warn'*, *err_text=''*, *fix_text='Fixed.'*, *fix=None*, *fixable=True*)
Execute the verification with selected option.

scale(*type=None*, *option='old'*, *bscale=1*, *bzero=0*)
Scale image data by using `BSCALE` and `BZERO`.

Calling this method will scale `self.data` and update the keywords of `BSCALE` and `BZERO` in `self._header` and `self._image_header`. This method should only be used right before writing to the output file, as the data will be scaled and is therefore not very usable after the call.

Parameters

type : str, optional

destination data type, use a string representing a numpy dtype name, (e.g. `'uint8'`, `'int16'`, `'float32'` etc.). If is `None`, use the current data type.

option : str, optional

how to scale the data: if `"old"`, use the original `BSCALE` and `BZERO` values when the data was read/created. If `"minmax"`, use the minimum and maximum of the data to scale. The option will be overwritten by any user-specified bscale/bzero values.

bscale, bzero : int, optional

user specified `BSCALE` and `BZERO` values.

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 are 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 are 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).

columns
The ColDefs objects describing the columns in this table.

shape
Shape of the image array–should be equivalent to self.data.shape.

size
Size (in bytes) of the data portion of the HDU.

2.7.3 Section

class pyfits.**Section**(*hdu*)
Bases: object

Image section.

Slices of this object load the corresponding section of an image array from the underlying FITS file on disk, and applies any BSCALE/BZERO factors.

Section slices cannot be assigned to, and modifications to a section are not saved back to the underlying file.

See the *Data Sections* section of the PyFITS documentation for more details.

2.8 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.

2.8.1 FITSDiff

class pyfits.**FITSDiff**(*a*, *b*, *ignore_keywords=[]*, *ignore_comments=[]*, *ignore_fields=[]*, *numdiffs=10*, *tolerance=0.0*, *ignore_blanks=True*, *ignore_blank_cards=True*)
Bases: pyfits.diff._BaseDiff

Diff two FITS files by filename, or two HDUList objects.

FITSDiff objects have the following diff attributes:

- diff_hdu_count: If the FITS files being compared have different numbers of HDUs, this contains a 2-tuple of the number of HDUs in each file.
- diff_hdus: If any HDUs with the same index are different, this contains a list of 2-tuples of the HDU index and the HDUDiff object representing the differences between the two HDUs.

Parameters

a : str or HDUList

The filename of a FITS file on disk, or an HDUList object.

b : str or HDUList

The filename of a FITS file on disk, or an HDUList object to compare to the first file.

ignore_keywords : sequence, optional

Header keywords to ignore when comparing two headers; the presence of these keywords and their values are ignored. Wildcard strings may also be included in the list.

ignore_comments : sequence, optional

A list of header keywords whose comments should be ignored in the comparison. May contain wildcard strings as with ignore_keywords.

ignore_fields : sequence, optional

The (case-insensitive) names of any table columns to ignore if any table data is to be compared.

numdiffs : int, optional

The number of pixel/table values to output when reporting HDU data differences. Though the count of differences is the same either way, this allows controlling the number of different values that are kept in memory or output. If a negative value is given, then numdifs is treated as unlimited (default: 10).

tolerance : float, optional

The relative difference to allow when comparing two float values either in header values, image arrays, or table columns (default: 0.0).

ignore_blanks : bool, optional

Ignore extra whitespace at the end of string values either in headers or data. Extra leading whitespace is not ignored (default: True).

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
['*']
```

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

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.

2.8.2 HDUDiff

class pyfits.**HDUDiff**(*a*, *b*, *ignore_keywords=[]*, *ignore_comments=[]*, *ignore_fields=[]*, *numdiffs=10*, *tolerance=0.0*, *ignore_blanks=True*, *ignore_blank_cards=True*)
Bases: `pyfits.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).

See `FITSDiff` for explanations of the initialization parameters.

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
['*']
```

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

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.

2.8.3 HeaderDiff

class pyfits.**HeaderDiff**(*a*, *b*, *ignore_keywords=[]*, *ignore_comments=[]*, *tolerance=0.0*, *ignore_blanks=True*, *ignore_blank_cards=True*)

Bases: `pyfits.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.

See `FITSDiff` for explanations of the initialization parameters.

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
['*']
```

`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

`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.

2.8.4 `ImageDataDiff`

class `pyfits.`**`ImageDataDiff`** (*a*, *b*, *numdiffs=10*, *tolerance=0.0*)

Bases: `pyfits.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.

See `FITSDiff` for explanations of the initialization parameters.

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
['*']
```

`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

`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.

2.8.5 `RawDataDiff`

class `pyfits.`**`RawDataDiff`** (*a*, *b*, *numdiffs=10*)
Bases: `pyfits.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.

See FITSDiff for explanations of the initialization parameters.

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
['*']
```

`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

`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.

2.8.6 `TableDataDiff`

class `pyfits.`**`TableDataDiff`** (*a*, *b*, *ignore_fields=[]*, *numdiffs=10*, *tolerance=0.0*)

Bases: `pyfits.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.

See `FITSDiff` for explanations of the initialization parameters.

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
['*']
```

`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

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.

2.9 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.

2.9.1 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.

2.9.2 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.

2.9.3 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 PyFITS will not take such action to hurt the integrity of the data.

Not all fixes may be the "correct" fix, but at least PyFITS will try to make the fix in such a way that it will not throw off other FITS readers.

2.9.4 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.

2.9.5 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.

CHAPTER 3

PyFITS Developers Guide

This "developers guide" will be brief, as PyFITS will, in the near future, be deprecated in favor of Astropy (which includes a port of PyFITS now dubbed `astropy.io.fits`. As such, it should be sufficient for any developers wishing to contribute to PyFITS to look at the developer documentation for Astropy, as much of it applies equally well. In particular, please look at the Astropy Coding Guidelines and the Documentation Guidelines before getting started with any major contributions to PyFITS (don't worry if you don't immediately absorb *everything* in those guidelines–it's just good to be aware that they exist and have a rough understanding of how to approach the source code).

3.1 Getting the source code

PyFITS was originally developed in SVN, but now most development has moved to Git, primarily for ease of syncing changes to Astropy. That said, the SVN repository is still maintained for legacy purposes. PyFITS' lead maintainer at STScI will handle synchronizing the Git and SVN repositories, but the steps for configuring git-svn are documented below for posterity. Outside users wishing to contribute to the source code should start with Astropy's guide to Contributing to Astropy.

The official PyFITS GitHub page is at: https://github.com/spacetelescope/pyfits

The best way to contribute to PyFITS is to create an account on GitHub, fork your own copy of the PyFITS repository, and then make your changes in your personal fork and make a pull request when they are ready to share. The entire process is described in Astropy's Workflow for Developers document. That documention was written for Astropy, but applies all the same to PyFITS. Just replace any instance of `astropy/astropy.git` with `spacetelescope/PyFITS.git` and so on. Use of virtualenv and `./setup.py develop` are strongly encouraged for developing on PyFITS–use of this tools is also described in the aforementioned Workflow for Developers document.

3.1.1 Synchronizing with SVN

This section is primarily intended for developers at STScI who have commit access to the PyFITS SVN repository (http://svn6.assembla.com/svn/pyfits). The PyFITS Git and SVN repositories are synced using the git-svn command. git-svn can be tricky to install as it requires the Perl bindings for SVN, as well as SVN itself and of course Git. The easiest way to get git-svn is to ask a system administrator to install it from the OS packaging system.

Most guides for setting up git-svn start out with either the `git svn init` command or `git svn clone`. But because the work of synchronizing the Git and SVN repositories up until this point has already been done, a faster, though seemingly less straightforward approach, is to just clone the GitHub reposiotry and add the git-svn metadata manually:

1. Clone the main spacetelescope GitHub repository:

   ```
   git clone git@github.com:spacetelescope/PyFITS.git
   ```

2. cd into the repository and open `.git/config` in an editor and add the following:

   ```
   [svn]
       authorsfile = .authors
   [svn-remote "svn"]
       url = http://svn6.assembla.com/svn/pyfits
       fetch = trunk:refs/remotes/trunk
       branches = branches/*:refs/remotes/branches/*
       tags = tags/*:refs/remotes/tags/*
   [branch "3.0-stable"]
       remote = .
       merge = refs/remotes/branches/3.0-stable
   [branch "3.1-stable"]
       remote = .
       merge = refs/remotes/branches/3.1-stable
   ```

   Repeat the `[branch "X.Y-stable"]` section following the above pattern for any actively maintained release branches (see the "Maintenance" section below for more details on release branches).

   Warning: Do not forget to set the `[svn]/authorsfile = .authors` option, or the repository will get severely confused when trying to sync SVN changes with the git repository. The .authors file maps SVN usernames to developers' name/e-mail address to use in git commits. If you intend to synchronize changes you make with SVN, make sure to add yourself to the .authors file. The format should be self-explanatory.

3. Put the hash of the latest revision of the upstream master branch in refs file for trunk, so git-svn knows where to start synchronizing with SVN's trunk:

   ```
   git rev-parse origin/master > .git/refs/remotes/trunk
   ```

4. Finally, do:

   ```
   git svn fetch
   ```

   to synchronize any new revisions in the SVN repository.

Syncing new changes to SVN

The command for committing new changes in git to the SVN repository is `git svn dcommit`. This command goes through all commits on the current branch that have *not* yet been committed to SVN and does so.

Whenever you are about to push new changes on the master branch to the remote remote repository on GitHub it is best to first cross-commit those changes to SVN. This is because git-svn rewrites the commit messages on all your commits to include a reference to the SVN revision that was created from that commit. So if you push first, and then run `git svn dcommit` you will now have different commits (as far as their SHA has is concerned) on your local repository from what you just pushed to the remote repository. The simplest way to resolve this, when it happens, is to `git push --force`. This will override the old history with the new history that includes the SVN revisions in the commit messages.

It's easier, however, to remember to always run `git svn dcommit` before doing a `git push`.

3.2 Maintenance

At any given time there are two to three lines of development on PyFITS (possibly more if some critical bug is discovered that needs to be backported to older release lines, though such situations are rare). Typically there is the mainline development in the 'master' branch, and at least one branch named after the last minor release. For example, if the version being developed in the mainline is '3.2.0' there will be, at a minimum, a '3.1-stable' branch into which bug fixes can be ported. There may also be a '3.0-stable' branch and so on so long as new bugfix releases are being made with '3.0.z' versions.

Bug fix releases should never add new public APIs or change existing ones–they should only correct bugs or major oversights. "Minor" releases, where the second number in the version is increased, may introduce new APIs and may *deprecate* old interfaces (see the `@deprecated` decorated in `pyfits.util`, but may not otherwise remove or change (non-buggy) behavior of old interfaces without backwards compatibility with the previous versions in the same major version line. Major releases may break backwards compatibility so long as warning has been given through `@deprecated` markers and documentation that those interfaces will break in future versions.

In general all development should be done in the 'master' branch, including development of new features and bug fixes (though temporary branches should certainly be used aggressively for any individual feature or fix being developed, they should be merged back into 'master' when ready).

The only exception to this rule is when developing a bug fix that *only* applies to an older release line. For example it's possible for a bug to exist in version '3.1.1' that no longer exists in the 'master' branch (perhaps because it pertains to an older API), but that still exists in the '3.1-stable' branch. Then that bug should be fixed in the '3.1-stable' branch to be included in the version '3.1.2' bugfix release (assuming a bugfix release is planned). If that bug pertains to any older release branches (such as '3.0-stable') it should also be backported to those branches by way of `git cherry-pick`.

3.3 Releasing

Creating a PyFITS release consists 3 main stages each with several sub-steps according to this rough outline:

1. Pre-release
   (a) Set the version string for the release in the setup.cfg file
   (b) Set the release date in the changelog (CHANGES.txt)
   (c) Test that README.txt and CHANGES.txt can be correctly parsed as RestructuredText.
   (d) Commit these preparations to the repository, creating a specific commit to tag as the "release"
2. Release
   (a) Create a tag from the commit created in the pre-release stage
   (b) Register the new release on PyPI
   (c) Build a source distribution of the release and test that it is installable (specifically, installable with pip) and that all the tests pass from an installed version
3. Post-release
   (a) Upload the source distribution to PyPI
   (b) Set the version string for the "next" release in the setup.cfg file (the choice of the next version is based on inference, and does not mean the "next" version can't be changed later if desired)
   (c) Create a new section in CHANGES.txt for the next release (using the same "next" version as in part b)
   (d) Commit these "post-release" changes to the repository
   (e) Push the release commits and the new tag to the remote repository (GitHub)

(f) Update the PyFITS website to reflect the new version

(g) Build Windows installers for all supported Python versions and upload them to PyPI

Most of these steps are automated by using zest.releaser along with some hooks designed specifically for PyFITS that automate actions such as updating the PyFITS website.

3.3.1 Prerequisites for performing a release

1. Because PyFITS is released (registered and uploaded to) on PyPI it is necessary to create an account on PyPI and get assigned a "Maintainer" role for the PyFITS package. Currently the package owners–the only two people who can add additional Maintainers are Erik Bray <embray@stsci.edu> and Nicolas Barbey <nicolas.a.barbey@gmail.com>. (It remains a "todo" item to add a shared "space telescope" account. In the meantime, should both of those people be hit by a bus simultaneously the PyPI administrators will be reasonable if the situation is explained to them with proper documentation).

   Once your PyPI account is set up, it is necessary to add your PyPI credentials (username and password) to the `.pypirc` file in your home directory with the following format:

   ```
   [server-login]
   username: <your PyPI username>
   password: <your PyPI password>
   ```

   Unfortunately some the `setup.py` commands for interacting with PyPI are broken in that they don't allow interactive password entry. Creating the `.pypirc` file is *currently* the most reliable way to make authentication with PyPI "just work". Be sure to `chmod 600` this file.

2. Generate a signing key–all PyFITS tags are now cryptographically signed when creating the tag (using `git tag -s`). The Astropy release process page documents how to set this up.

3. Also make sure to have an account on readthedocs.org with administrative access to the PyFITS project on Read the Docs: https://readthedocs.org/projects/pyfits/ This hosts documentation for all (recent) versions of PyFITS. (TODO: Here also we need a "space telescope" account with administrative rights to all STScI projects that use RtD.)

4. It's best to do the release in a relatively "clean" Python environment, so make sure you have virtualenv installed and that you've had some practice in using it.

5. Make sure you have Numpy and nose installed and are able to run the PyFITS tests successfully without any errors. Even better if you can do this with tox.

6. Make sure that at least someone can make the Windows builds. This requires a Windows machine with at least Windows XP, Mingw32 with msys, and all of the Python development packages. Python versions 2.5, 2.6, 2.7, 3.1, and 3.2 should be installed with the installers from python.org, as well as a recent version of Numpy for each of those Python versions (currently Numpy 1.6.x), as well as Git. (TODO: More detailed instructions for setting up a Windows development environment.)

7. PyFITS also has a page on STScI's website: http://www.stsci.edu/institute/software_hardware/pyfits. This is normally the first hit when Googling 'pyfits' so it's important to keep up to date. At a minimum each release should update the front page to mention the most recent release, the Release Notes page with an HTML rendering of the most recent changelog, and the download page with links to all the current versions. See the exisint site for examples. The STScI website has both a test server and a production server. It's difficult for content creators to get direct access to the production server, but at least make sure you have access to the test server on port 8072, and that IT has given you permission to write to the PyFITS section of the site.

   Part of the PyFITS automated release script attempts to update the PyFITS website (on the test server) as part of the standard release process. So it's important to test your access to the site and ability to make edits. If for any reason the automatic update fails (e.g. your authentication fails) it is still possible to update the site manually.

Once the updates are made it's necessary to have IT push the updates to the production server. As of writing the best person to ask is George Smyth– asking him directly is the fastest way to get it done, though if you send a ticket to IT it will be handled eventually.

8. Triage issues is milestones in the PyFITS bug tracker(s). Currently this includes the Trac site: https://trac.assembla.com/pyfits/roadmap and the GitHub site: https://github.com/spacetelescope/PyFITS/issues/milestones

   No new tickets are being added in Trac, so after all open tickets in the Trac site have been addressed, milestones will only need to be managed in GitHub.

   First create a new milestone for the version after the version to be released. If a major/minor release is being made, make the milestone for the next bugfix release in that series as well. For example if releasing a bugfix release like 3.0.1, create a milestone for 3.0.2. If releasing 3.1.0, create milestones for 3.2.0 *and* 3.2.1.

   If the milestone for the to be released version still has any issues remaining in it, such as bugs that were not fixed, move them to the next appriopriate milestone if they will not be addressed before the release (or close issues that are no longer applicable). Ensure that the milestone for the to be released version has no open issues remaining in it.

3.3.2 Release procedure

(These instructions are adapted from the Astropy release process which itself was adapted from PyFITS' release process–the former just got written down first.)

1. In a directory outside the pyfits repository, create an activate a virtualenv in which to do the release (it's okay to use `--system-site-packages` for dependencies like Numpy):

   ```
   $ virtualenv --system-site-packages --distribute pyfits-release
   $ source pyfits-release/bin/activate
   ```

2. Obtain a *clean* version of the PyFITS repository. That is, one where you don't have any intermediate build files. It is best to use a fresh `git clone` from the main repository on GitHub without any of the git-svn configuration. This is because the git-svn support in zest.releaser does not handle tagging in branches very well yet.

3. Use `git checkout` to switch to the appropriate branch from which to do the release. For a new major or minor release (such as 3.0.0 or 3.1.0) this should be the 'master' branch. When making a bugfix release it is necessary to switch to the appropriate bugfix branch (e.g. `git checkout 3.1-stable` to release 3.1.2 up from 3.1.1).

4. 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
   ```

5. Install `stsci.distutils` which includes some additional releaser hooks that are useful:

   ```
   $ pip install stsci.distutils --upgrade --force
   ```

6. Ensure that any lingering changes to the code have been committed, then start the release by running:

   ```
   $ fullrelease
   ```

7. 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.txt and setup.cfg showing that a release date has been added to the changelog, and that the version has been updated in setup.cfg. Enter 'Y' when asked to commit these changes.

8. You will then be shown the command that will be run to tag the release. Enter 'Y' to confirm and run the command.
9. When asked "Check out the tag (for tweaks or pypi/distutils server upload)" enter 'Y': This feature is used when uploading the source distribution to our local package index. When asked to 'Register and upload' to PyPI enter 'N'. We will do this manually later in the process once we've tested the release out first. If asked to add the package to the "STScI package index" enter 'N'–this package index is no longer being maintained.
10. 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 "3.1.0" the default next version will be "3.1.1". If we want the next version to be, say "3.2.0" then that must be entered manually.
11. You will be shown a diff of CHANGES.txt 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.
12. When asked to push the changes to a remote repository, enter 'N'. We want to test the release out before pushing changes to the remote repository or registering in PyPI.
13. When asked to update the PyFITS homepage enter 'Y'. The enter the name of the previous version (in the same MAJOR.MINOR.x branch) and then the name of the just released version. The defaults will usually be correct. When asked, enter the username and password for your Zope login. As of writing this is not necessarily the same as your Exchange password. If the update succeeeds make sure to e-mail IT and ask them to push the updated pages from the test site to the production site.

    This should complete the portion of the process that's automated at this point (though future versions will automate these steps as well, after a few needed features are added to zest.releaser).
14. Check out the tag of the released version. For example:

    ```
    $ git checkout v3.1.0
    ```
15. Create the source distribution by doing:

    ```
    $ python setup.py sdist
    ```
16. Now, outside the repository create and activate another new virtualenv for testing the release:

    ```
    $ virtualenv --system-site-packages --distribute pyfits-release-test
    $ source pyfits-release-test/bin/activate
    ```
17. Use `pip` to install the source distribution built in step 13 into the new test virtualenv. This will look something like:

    ```
    $ pip install PyFITS/dist/pyfits-3.2.0.tar.gz
    ```

    where the path should be to the sole `.tar.gz` file in the `dist/` directory under your repository clone.
18. Try running the tests in the installed PyFITS:

    ```
    $ pip install nose --force --upgrade
    $ nosetests pyfits
    ```

    If any of the tests fail abort the process and start over. Undo the previous git commit (where you bumped the version):

    ```
    $ git reset --hard HEAD^
    ```

    Resolve the test failure, commit any new fixes, and start the release procedure over again (it's rare for this to be an issue if the tests passed *before* starting the release, but it is possible–the most likely case being if some

file that *should* be installed is either not getting installed or is not included in the source distribution in the first place).

19. Assuming the test installation worked, change directories back into the repository and push the new tag/release to the main repository on GitHub:

    ```
    $ git push --tags
    ```

    This initial step is necessary since the tag was made off of a pure git commit. But when we synchronize with SVN the commit history will change so we need to force an additional push to the GitHub repository:

    ```
    $ git svn dcommit
    $ git push --force
    ```

    Then register the release on PyPI with:

    ```
    $ python setup.py register
    ```

    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
    ```

    After registering on PyPI go to the URL:

    https://pypi.python.org/pypi?%3Aaction=pkg_edit&name=pyfits

    and mark any previous releases superceded by this release as hidden via the web UI. Don't check "Auto-hide old releases" as we want to support discovery of bugfix releases of older versions.

20. When releasing a new major or minor version, create a bugfix branch for that version. Starting from the tagged changset, just checkout a new branch and push it to the remote server. For example, after releasing version 3.2.0, do:

    ```
    $ git checkout -b 3.2-stable
    ```

    Then edit the setup.cfg so that the version is `'3.2.1.dev'`, and commit that change. Then, do:

    ```
    $ git push origin +3.2-stable
    ```

    Note: You may need to replace `origin` here with `upstream` or whatever remote name you use for the main PyFITS repository on GitHub.

    The purpose of this branch is for creating bugfix releases like "3.2.1" and "3.2.2", 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 bugfix branches.

21. On the other hand, if a bugfix release was made, the `CHANGES.txt` file will only be updated in the stable branch; the master branch also needs to be updated so that the release is reflected in its copy of `CHANGES.txt`. Just run:

    ```
    $ git checkout master
    ```

    Say 3.2.1 was just released. Use `git log -p` to find the commit that updated the changelog with the release date in the stable branch, like:

    ```
    $ git log -p 3.2-stable
    ```

    Copy the commit hash, and then cherry-pick it into master:

```
$ git cherry-pick <sha1 hash>
```

You will likely have to resolve a merge conflict, but just make sure that the section heading for the just released version is updated so that "(unreleased)" is replaced with today's date. Also ensure that a new section is added for the next bugfix release in that release series.

22. Log into the Read the Docs control panel for PyFITS at https://readthedocs.org/projects/pyfits/. Click on "Admin" and then "Versions". Find the just-released version (it might not appear for a few minutes) and click the check mark next to "Active" under that version. Leave the dropdown list on "Public", then scroll to the bottom of the page and click "Submit". If this is the release with the highest version number, make sure to set it as the "default" version as soon as the build finishes.

    Note: When you first activate the new version in Read the Docs, it immediately displays a "Build Failed" message for the build of the new docs. This is a bug–all it really means is that those docs have never been built yet. Give it a few minutes before checking that the build succeeded. Then you can set that version as the default if needed.

23. We also mirror the most recent documentation at pythonhosted.org/pyfits (formerly packages.python.org).

    First it is necessary to build the docs manually. Make sure all the dependencies are satisfied by running:

    ```
    $ pip install sphinx
    ```

    Then change directories into the docs/ directory and install the additional requirements for the docs:

    ```
    $ cd docs
    $ pip install -r requirements.txt
    ```

    Then make the HTML docs:

    ```
    make html
    ```

    Now change directories back to the source root and upload:

    ```
    $ cd ..
    $ python setup.py upload_docs
    ```

24. Mark the milestone of the released version as closed/completed in the PyFITS bug tracker(s). If asked for a timestamp (as Trac does) use the timestamp of the git tag made for the release.

25. Build and upload the Windows installers:

    (a) Launch a MinGW shell.

    (b) Just as before make sure you have a `pypirc` file in your home directory with your authentication info for PyPI. On Windows the file should be called just `pypirc` without the leading `.` because having some consistency would make this too easy :)

    (c) Do a `git clone` of the repository or, if you already have a clone of the repository do `git fetch --tags` to get the new tags.

    (d) Check out the tag for the just released version. For example:

    ```
    $ git checkout v3.2.0
    ```

    (ignore the message about being in "detached HEAD" state).

    (e) For each Python version installed, build with the mingw32 compiler, create the binary installer, and upload it. It's best to use the full path to each Python version to avoid ambiguity. It is also best to clean the repository between builds for each version. For example:

```
$ /C/Python25/python setup.py build -c mingw32 bdist_wininst upload
< ... builds and uploads successfully ... >
$ git clean -dfx
$ /C/Python26/python setup.py build -c mingw32 bdist_wininst upload
< ... builds and puloads successfully ... >
$ git clean -dfx
$ < ... and so on, for all currently supported Python versions ... >
```

CHAPTER 4

Appendix

4.1 Header Interface Transition Guide

PyFITS v3.1 included an almost complete rewrite of the `Header` interface. Although the new interface is largely compatible with the old interace (whether due to similarities in the design, or backwards-compatibility support), there are enough differences that a full explanation of the new interface is merited.

The Trac ticket discussing the initial motivation and changes to be made to the `Header` class is #64. It may be worth reading for some of the background to this work, though this document contains a more complete description of the "final" product (which will continue to evolve).

4.1.1 Background

Prior to 3.1, PyFITS users interacted with FITS headers by way of three different classes: `Card`, `CardList`, and `Header`.

The Card class represents a single header card with a keyword, value, and comment. It also contains all the machinery for parsing FITS header cards, given the 80 character string, or "card image" read from the header.

The CardList class is actually a subclass of Python's `list` built-in. It was meant to represent the actual list of cards that make up a header. That is, it represents an ordered list of cards in the physical order that they appear in the header. It supports the usual list methods for inserting and appending new cards into the list. It also supports `dict`-like keyword access, where `cardlist['KEYWORD']` would return the first card in the list with the given keyword.

A lot of the functionality for manipulating headers was actually buried in the CardList class. The Header class was more of a wrapper around CardList that added a little bit of abstraction. It also implemented a partial dict-like interface, though for Headers a keyword lookup returned the header value associated with that keyword, and not the Card object. Though almost every method on the Header class was just performing some operations on the underlying CardList.

The problem is that there were certain things one could *only* do by directly accessing the CardList, such as look up the comments on a card, or access cards that have duplicate keywords, such as HISTORY. Another long-standing misfeature that slicing a Header object actually returned a CardList object, rather than a new Header. For all but the most simple use cases, working with CardList objects was largely unavoidable.

But it was realized that CardList is really an implementation detail not representing any element of a FITS file distinct from the header itself. Users familiar with the FITS format know what a header is, but it's not clear how a "card list" is distinct from that, or why operations go through the Header object, while some have to be performed through the CardList.

So the primary goal of this redesign was eliminate the `CardList` class altogether, and make it possible for users to perform all header manipulations directly through `Header` objects. It also tries to present headers as similar as possible to more a more familiar data structure–an ordered mapping (or `OrderedDict` in Python) for ease of use

by new users less familiar with the FITS format. Though there are still many added complexities for dealing with the idiosyncracies of the FITS format.

4.1.2 Deprecation Warnings

A few old methods on the `Header` class have been marked as deprecated, either because they have been renamed to a more PEP 8-compliant name, or because have become redundant due to new features. To check if your code is using any deprecated methods or features, run your code with `python -Wd`. This will output any deprecation warnings to the console.

Two of the most common deprecation warnings related to Headers are for:

- `Header.has_key()`–this has actually been deprecated since PyFITS 3.0, just as Python's `dict.has_key` is deprecated. For checking a key's presence in a mapping object like `dict` or `Header`, use the `key in d` syntax. This has long been the preference in Python.
- `Header.ascardlist()` and `Header.ascard`–these were used to access the `CardList` object underlying a header. They should still work, and return a skeleton CardList implementation that should support most of the old CardList functionality. But try removing as much of this as possible. If direct access to the `Card` objects making up a header is necessary, use `Header.cards`, which returns an iterator over the cards. More on that below.

4.1.3 New Header Design

The new `Header` class is designed to work as a drop-in replacement for a `dict` via duck typing. That is, although it is not a subclass of `dict`, it implements all the same methods and interfaces. In particular, it is similar to an `OrderedDict` in that the order of insertions is preserved. However, Header also supports many additional features and behaviors specific to the FITS format. It should also be noted that while the old Header implementation also had a dict-like interface, it did not implement the *entire* dict interface as the new Header does.

Although the new Header is used like a dict/mapping in most cases, it also supports a `list` interface. The list-like interace is a bit idiosyncratic in that in some contexts the Header acts like a list of values, in some like a list of keywords, and in a few contexts like a list of `Cards`. This may be the most difficult aspect of the new design, but there is logic to it.

As with the old Header implementation, integer index access is supported: `header[0]` returns the value of the first keyword. However, the `Header.index()` method treats the header as though it's a list of keywords, and returns the index of a given keyword. For example:

```
>>> header.index('BITPIX')
2
```

`Header.count()` is similar to `list.count`, and also takes a keyword as its argument:

```
>>> header.count('HISTORY')
20
```

A good rule of thumb is that any item access using square brackets `[]` returns *value* in the header, whether using keyword or index lookup. Methods like `index()` and `count()` that deal with the order and quantity of items in the Header generally work on keywords. Finally, methods like `insert()` and `append()` that add new items to the header work on cards.

Aside from the list-like methods, the new Header class works very similarly to the old implementation for most basic use cases and should not present too many surprises. There are differences, however:

- As before, the Header() initializer can take a list of `Card` objects with which to fill the header. However, now any iterable may be used. It is also important to note that *any* Header method that accepts `Card` objects can

also accept 2-tuples or 3-tuples in place of Cards. That is, either a `(keyword, value, comment)` tuple or a `(keyword, value)` tuple (comment is assumed blank) may be used anywhere in place of a Card object. This is even preferred, as it simply involves less typing. For example:

```
>>> header = pyfits.Header([('A', 1), ('B', 2), ('C', 3, 'A comment')])
>>> header
A       =                    1
B       =                    2
C       =                    3 / A comment
```

- As demonstrated in the previous example, the `repr()` for a Header, that is the text that is displayed when entering a Header object in the Python console as an expression, shows the header as it would appear in a FITS file. This inserts newlines after each card so that it is easily readable regardless of terminal width. It is *not* necessary to use `print header` to view this. Simply entering `header` displays the header contents as it would appear in the file (sans the END card).
- `len(header)` is now supported (previously it was necessary to do `len(header.ascard)`. This returns the total number of cards in the header, including blank cards, but excluding the END card.
- FITS supports having duplicate keywords, although they are generally in error except for commentary keywords like COMMENT and HISTORY. PyFITS now supports reading, updating, and deleting duplicate keywords: Instead of using the keyword by itself, use a `(keyword, index)` tuple. For example `('HISTORY', 0)` represents the first HISTORY card, `('HISTORY', 1)` represents the second HISTORY card, and so on. In fact, when a keyword is used by itself, it's really just shorthand for `(keyword, 0)`. Its is now possible to delete an accidental duplicate like so:

  ```
  >>> del header[('NAXIS', 1)]
  ```

  This will remove an accdential duplicate NAXIS card from the header.
- Even if there are duplicate keywords, keyword lookups like `header['NAXIS']` will always return the value associated with the first copy of that keyword, with one exception: Commentary keywords like COMMENT and HISTORY are expected to have duplicates. So `header['HISTORY']`, for example, returns the whole sequence of HISTORY values in the correct order. This list of values can be sliced arbitrarily. For example, to view the last 3 history entries in a header:

  ```
  >>> hdulist[0].header['HISTORY'][-3:]
    reference table oref$laf13367o_pct.fits
    reference table oref$laf13369o_apt.fits
  Heliocentric correction = 16.225 km/s
  ```

- Subscript assignment can now be used to add new keywords to the header. Just as with a normal `dict`, `header['NAXIS'] = 1` will either update the NAXIS keyword if it already exists, or add a new NAXIS keyword with a value of `1` if it does not exist. In the old interface this would return a `KeyError` if NAXIS did not exist, and the only way to add a new keyword was through the update() method.

  By default, new keywords added in this manner are added to the end of the header, with a few FITS-specific exceptions:

  - If the header contains extra blank cards at the end, new keywords are added before the blanks.
  - If the header ends with a list of commentary cards–for example a sequence of HISTORY cards–those are kept at the end, and new keywords are inserted before the commentary cards.
  - If the keyword is a commentary keyword like COMMENT or HISTORY (or an empty string for blank keywords), a *new* commentary keyword is always added, and appended to the last commentary keyword of the same type. For example, HISTORY keywords are always placed after the last history keyword:

    ```
    >>> header = pyfits.Header()
    >>> header['COMMENT'] = 'Comment 1'
    ```

```
>>> header['HISTORY'] = 'History 1'
>>> header['COMMENT'] = 'Comment 2'
>>> header['HISTORY'] = 'History 2'
>>> header
COMMENT Comment 1
COMMENT Comment 2
HISTORY History 1
HISTORY History 2
```

These behaviors represent a sensible default behavior for keyword assignment, and represents the same behavior as `update()` in the old Header implementation. The default behaviors may still be bypassed through the use of other assignment methods like `Header.set()` and `Header.append()` described later.

- It is now also possible to assign a value and a comment to a keyword simultaneously using a tuple:

```
>>> header['NAXIS'] = (2, 'Number of axis')
```

This will update the value and comment of an existing keyword, or add a new keyword with the given value and comment.

- There is a new `Header.comments` attribute which lists all the comments associated with keywords in the header (not to be confused with COMMENT cards). This allows viewing and updating the comments on specific cards:

```
>>> header.comments['NAXIS']
Number of axis
>>> header.comments['NAXIS'] = 'Number of axes'
>>> header.comments['NAXIS']
Number of axes
```

- When deleting a keyword from a header, don't assume that the keyword already exists. In the old Header implementation this would just silently do nothing. For backwards-compatibility it is still okay to delete a non-existent keyword, but a warning will be raised. In the future this *will* be changed so that trying to delete a non-existent keyword raises a `KeyError`. This is for consistency with the behavior of Python dicts. So unless you know for certain that a keyword exists before deleting it, it's best to do something like:

```
>>> try:
...     del header['BITPIX']
... except KeyError:
...     pass
```

Or if you prefer to look before you leap:

```
>>> if 'BITPIX' in header:
...     del header['BITPIX']
```

- `del header` now supports slices. For example, to delete the last three keywords from a header:

```
>>> del header[-3:]
```

- Two headers can now be compared for equality–previously no two Header objects were the same. Now they compare as equal if they contain the exact same content. That is, this requires strict equality.
- Two headers can now be added with the '+' operator, which returns a copy of the left header extended by the right header with `extend()`. Assignment addition is also possible.
- The Header.update() method used commonly with the old Header API has been renamed to `Header.set()`. The primary reason for this change is very simple: Header implements the `dict` interface, which already has a method called update(), but that behaves differently from the old Header.update().

The details of the new update() can be read in the API docs, but it is very similar to `dict.update`. It also supports backwards compatibility with the old update() by analysis of the arguments passed to it, so existing code will not break immediately. However, this *will* cause a deprecation warning to be output if they're enabled. It is best, for starters, to replace all update() calls with set(). Recall, also, that direct assignment is now possible for adding new keywords to a header. So by and large the only reason to prefer using `Header.set()` is its capability of inserting or moving a keyword to a specific location using the `before` or `after` arguments.

- Slicing a Header with a slice index returns a new Header containing only those cards contained in the slice. As mentioned earlier, it used to be that slicing a Header returned a card list–something of a misfeature. In general, objects that support slicing ought to return an object of the same type when you slice them.

  Likewise, wildcard keywords used to return a CardList object. Now they return a new Header–similarly to a slice. For example:

  ```
  >>> header['NAXIS*']
  ```

  returns a new header containing only the NAXIS and NAXISn cards from the original header.

4.1.4 Transition Tips

The above may seem like a lot, but the majority of existing code using PyFITS to manipulate headers should not need to be updated, at least not immediately. The most common operations still work the same.

As mentioned above, it would be helpful to run your code with `python -Wd` to enable deprecation warnings–that should be a good idea of where to look to update your code.

If your code needs to be able to support older versions of PyFITS simultaneously with PyFITS 3.1, things are slightly trickier, but not by much–the deprecated interfaces will not be removed for several more versions because of this.

- The first change worth making, which is supported by any PyFITS version in the last several years, is remove any use of `Header.has_key()` and replace it with `keyword in header` syntax. It's worth making this change for any dict as well, since `dict.has_key` is deprecated. Running the following regular expression over your code may help with most (but not all) cases:

  ```
  s/([^ ]+)\.has_key\(([^)]+)\)/\2 in \1/
  ```

- If possible, replace any calls to Header.update() with Header.set() (though don't bother with this if you need to support older PyFITS versions). Also, if you have any calls to Header.update() that can be replaced with simple subscript assignments (eg. `header['NAXIS'] = (2, 'Number of axes')`) do that too, if possible.
- Find any code that uses `header.ascard` or `header.ascardlist()`. First ascertain whether that code really needs to work directly on Card objects. If that is definitely the case, go ahead and replace those with `header.cards`–that should work without too much fuss. If you do need to support older versions, you may keep using `header.ascard` for now.
- In the off chance that you have any code that slices a header, it's best to take the result of that and create a new Header object from it. For example:

  ```
  >>> new_header = pyfits.Header(old_header[2:])
  ```

  This avoids the problem that in PyFITS <= 3.0 slicing a Header returns a CardList by using the result to initialize a new Header object. This will work in both cases (in PyFITS 3.1, initializing a Header with an existing Header just copies it, a la `list`).
- As mentioned earlier, locate any code that deletes keywords with `del`, and make sure they either look before they leap (`if keyword in header:`) or ask forgiveness (`try/except KeyError:`).

Other Gotchas

- As mentioned above it is not necessary to enter `print header` to display a header in an interactive Python prompt. Simply entering `>>> header` by itself is sufficient. Using `print` usually will *not* display the header readably, because it does not include line-breaks between the header cards. The reason is that Python has two types of string representations: One is returned when one calls `str(header)` which happens automatically when you `print` a variable. In the case of the Header class this actually returns the string value of the header as it is written literally in the FITS file, which includes no line breaks.

  The other type of string representation happens when one calls `repr(header)`. The `repr` of an object is just meant to be a useful string "representation" of the object; in this case the contents of the header but with linebreaks betwen the cards and with the END card and padding trailing padding stripped off. This happens automatically when one enters a variable at the Python prompt by itself without a `print` call.

- The current version of the FITS Standard (3.0) states in section 4.2.1 that trailing spaces in string values in headers are not significant and should be ignored. PyFITS < 3.1 *did* treat treat trailing spaces as significant. For example if a header contained:

  KEYWORD1= 'Value '

  then `header['KEYWORD1']` would return the string `'Value '` exactly, with the trailing spaces intact. The new Header interface fixes this by automatically stripping trailing spaces, so that `header['KEYWORD1']` would return just `'Value'`.

  There is, however, one convention used by the IRAF ccdmosiac task for representing its TNX World Coordinate System and ZPX World Coordinate System non-standard WCS' that uses a series of keywords in the form `WATj_nnn` which store a text description of coefficients for a non-linear distortion projection. It uses its own microformat for listing the coefficients as a string, but the string is long, and thus broken up into several of these `WATj_nnn` keywords. Correct recombination of these keywords requires treating all whitespace literally. This convention either overlooked or predated the prescribed treatment of whitespace in the FITS standard.

  To get around this issue a global variable `pyfits.STRIP_HEADER_WHITESPACE` was introduced. Temporarily setting `pyfits.STRIP_HEADER_WHITESPACE = False` before reading keywords affected by this issue will return their values with all trailing whitespace intact.

  A future version of PyFITS may be able to detect use of conventions like this contextually and behave according to the convention, but in most cases the default behavior of PyFITS is to behave according to the FITS Standard.

4.2 PyFITS FAQ

Contents

4.2.1 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.

4.2.2 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.

See below for a few answers to other specific questions about Windows installation. For other installation errors not mentioned by this FAQ, please contact help@stsci.edu with a description of the problem.

On Windows Vista or later why can't the installer find Python in the registry?

This is a common issue with Windows installers for Python packages that do not support the new User Access Control (UAC) framework added in Windows Vista and later. In particular, when a Python is installed "for all users" (as opposed to for a single user) it adds entries for that Python installation under the `HKEY_LOCAL_MACHINE` (HKLM) hierarchy and *not* under the `HKEY_CURRENT_USER` (HKCU) hierarchy. However, depending on your UAC settings, if the PyFITS installer is not executed with elevated privileges it will not be able to check in HKLM for the required information about your Python installation.

In short: If you encounter this problem it's because you need the appropriate entries in the Windows registry for Python. You can download this script and execute it with the same Python as the one you want to install PyFITS into. For example to add the missing registry entries to your Python 2.7:

```
C:\>C:\Python27\python.exe C:\Path\To\Downloads\win_register_python.py
```

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
```

4.2.3 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.

4.3 Changelog

4.3.1 3.1.7 (unreleased)

- Nothing changed yet.

4.3.2 3.1.6 (2014-05-14)

- Nominal support for Python 3.4.
- Fixed a bug with using the `tabledump` and `tableload` functions with tables containing array columns (columns in which each element is an array instead of a single scalar value). (Backported from 3.2.3)
- Fixed an issue where PyFITS allowed newline characters in header values and comments. (Backported from 3.2.3)
- Fixed pickling of `FITS_rec` (table data) objects. (Backported from 3.2.3)
- Improved behavior when writing large compressed images on OSX by removing an unncessary check for platform architecture. (Backported from 3.2.3)
- Allow reading FITS files from file-like objects that do not have a `.closed` attribute (and as such may not even have an "open" vs. "closed" concept). (Backported from 3.2.3)
- Fixed minor issue with comparison of header commentary card values. (Backported from 3.2.3)

4.3.3 3.1.5 (2014-03-28)

- Fixed a regression on deletion of record-valued keyword cards using the Header wildcard syntax. This was intended to be fixed before the v3.1.4 release.

4.3.4 3.1.4 (2014-03-04)

- Added missing features from the `Header.insert()` method that were intended for inclusion in the original 3.1 release: In addition to accepting an integer index as the first argument, it also supports supplying a keyword name as the first argument for insertion relative to a specific keyword. It also now supports an optional `after` argument. If `after=True` the the insertion is made below the insertion point instead of above it. (Backported from 3.2.1)
- A grab bag of minor performance improvements in headers. (Backported from 3.2.1)
- Fixed an issue where opening an image containing pseudo-unsigned integers and immediately writing it to a new file using the `writeto` method would drop the scale factors that identified the data as unsigned. (Backported from 3.2.1)
- Fixed a bug where writing a file with `checksum=True` did not add the checksum on new files. (Backported from 3.2.1)
- Fixed an issue where validating an HDU's checksums removed the checksum from that HDU's header entirely (even if it was valid.) (Backported from 3.2.1)
- Fixed an issue where the size of the heap was sometimes not computed properly when writing an existing table containing variable-length array columns to a new FITS file. This could result in corruption in the new FITS file. (Backported from 3.2.1)
- Fixed a bug where a boolean value of `True` in a header could not be replaced with the integer 1, and likewise for `False` and 0 and vice versa. (Backported from 3.2.1)
- Fixed an issue similar to the above one but for numeric values–now replacing a header value with an equivalent numeric value will up/downcast that value. For example replacing '0' with '0.0' will write '0.0' to the header so that it is returned as a floating point value. Likewise a float can be downcast to an integer. (Backported from 3.2.1)
- Fixed unrelated crash when a header contains an invalid END card (for example "END = "). This resulted in a cryptic traceback. Now headers like this will detect "clearly intended" END cards and produce a warning about their invalidity and fix them. (Backported from 3.2.1)
- Fixed a display formatting issue with fitsdiff where sometimes it did not show the difference between two floating point numbers if they were the same up to some low number of digits. (Backported from 3.2.1)
- Fixed an issue where Python 2 sometimes allowed non-ASCII strings to be assigned as header values if they were assigned as old-style `str` objects and not `unicode` objects. (Backported from 3.2.1)

4.3.5 3.0.13 (2014-03-04)

- Fixed a bug where writing a file with `checksum=True` did not add the checksum on new files. (Backported from 3.2.1)
- Fixed an issue where validating an HDU's checksums removed the checksum from that HDU's header entirely (even if it was valid.) (Backported from 3.2.1)

4.3.6 3.2 (2013-11-26)

Highlights

- Rewrote CFITSIO-based backend for handling tile compression of FITS files. It now uses a standard CFITSIO instead of heavily modified pieces of CFITSIO as before. PyFITS ships with its own copy of CFITSIO v3.35 which supports the latest version of the Tiled Image Convention (v2.3), but system packagers may choose instead

to strip this out in favor of a system-installed version of CFITSIO. Earlier versions may work, but nothing earlier than 3.28 has been tested yet. (#169)

- Added support for reading and writing tables using the Q format for columns. The Q format is identical to the P format (variable-length arrays) except that it uses 64-bit integers for the data descriptors, allowing more than 4 GB of variable-length array data in a single table. (#160)
- Added initial support for table columns containing pseudo-unsigned integers. This is currently enabled by using the `uint=True` option when opening files; any table columns with the correct BZERO value will be interpreted and returned as arrays of unsigned integers.
- Some refactoring of the table and `FITS_rec` modules in order to better separate the details of the FITS binary and ASCII table data structures from the HDU data structures that encapsulate them. Most of these changes should not be apparent to users (but see API Changes below).

API Changes

- Assigning to values in `ColDefs.names`, `ColDefs.formats`, `ColDefs.nulls` and other attributes of `ColDefs` instances that return lists of column properties is no longer supported. Assigning to those lists will no longer update the corresponding columns. Instead, please just modify the `Column` instances directly (`Column.name`, `Column.null`, etc.)
- The `pyfits.new_table` function is marked "pending deprecation". This does not mean it will be removed outright or that its functionality has changed. It will likely be replaced in the future for a function with similar, if not subtly different functionality. A better, if not slightly more verbose approach is to use `pyfits.FITS_rec.from_columns` to create a new `FITS_rec` table–this has the same interface as `pyfits.new_table`. The difference is that it returns a plan `FITS_rec` array, and not an HDU instance. This `FITS_rec` object can then be used as the data argument in the constructors for `BinTableHDU` (for binary tables) or `TableHDU` (for ASCII tables). This is analogous to creating an `ImageHDU` by passing in an image array. `pyfits.FITS_rec.from_columns` is just a simpler way of creating a FITS-compatible recarray from a FITS column specification.
- The `updateHeader`, `updateHeaderData`, and `updateCompressedData` methods of the `CompDataHDU` class are pending deprecation and moved to internal methods. The operation of these methods depended too much on internal state to be used safely by users; instead they are invoked automatically in the appropriate places when reading/writing compressed image HDUs.
- The `CompDataHDU.compData` attribute is pending deprecation in favor of the clearer and more PEP-8 compatible `CompDataHDU.compressed_data`.
- The constructor for `CompDataHDU` has been changed to accept new keyword arguments. The new keyword arguments are essentially the same, but are in underscore_separated format rather than camelCase format. The old arguments are still pending deprecation.
- The internal attributes of HDU classes `_hdrLoc`, `_datLoc`, and `_datSpan` have been replaced with `_header_offset`, `_data_offset`, and `_data_size` respectively. The old attribute names are still pending deprecation. This should only be of interest to advanced users who have created their own HDU subclasses.
- The following previously deprecated functions and methods have been removed entirely: `createCard`, `createCardFromString`, `upperKey`, `ColDefs.data`, `setExtensionNameCaseSensitive`, `_File.getfile`, `_TableBaseHDU.get_coldefs`, `Header.has_key`, `Header.ascardlist`.

  If you run your code with a previous version of PyFITS (>= 3.0, < 3.2) with the `python -Wd` argument, warnings for all deprecated interfaces still in use will be displayed.
- Interfaces that were pending deprecation are now fully deprecated. These include: `create_card`, `create_card_from_string`, `upper_key`, `Header.get_history`, and `Header.get_comment`.

- The `.name` attribute on HDUs is now directly tied to the HDU's header, so that if `.header['EXTNAME']` changes so does `.name` and vice-versa.
- The `pyfits.file.PYTHON_MODES` constant dict was renamed to `pyfits.file.PYFITS_MODES` which better reflects its purpose. This is rarely used by client code, however. Support for the old name will be removed by PyFITS 3.4.

Other Changes and Additions

- The new compression code also adds support for the ZQUANTIZ and ZDITHER0 keywords added in more recent versions of this FITS Tile Compression spec. This includes support for lossless compression with GZIP. (#198) By default no dithering is used, but the `SUBTRACTIVE_DITHER_1` and `SUBTRACTIVE_DITHER_2` methods can be enabled by passing the correct constants to the `quantize_method` argument to the `CompImageHDU` constuctor. A seed can be manually specified, or automatically generated using either the system clock or checksum-based methods via the `dither_seed` argument. See the documentation for `CompImageHDU` for more details. (#198) (spacetelescope/PYFITS#32)
- Images compressed with the Tile Compression standard can now be larger than 4 GB through support of the Q format. (#159)
- All HDUs now have a `.ver` `.level` attribute that returns the value of the EXTVAL and EXTLEVEL keywords from that HDU's header, if the exist. This was added for consistency with the `.name` attribute which returns the EXTNAME value from the header.
- Then `Column` and `ColDefs` classes have new `.dtype` attributes which give the Numpy dtype for the column data in the first case, and the full Numpy compound dtype for each table row in the latter case.
- There was an issue where new tables created defaulted the values in all string columns to '0.0'. Now string columns are filled with empty strings by default–this seems a less surprising default, but it may cause differences with tables created with older versions of PyFITS.
- Improved round-tripping and preservation of manually assigned column attributes (`TNULLn`, `TSCALn`, etc.) in table HDU headers. (astropy/astropy#996)

Bug Fixes

- Binary tables containing compressed images may, optionally, contain other columns unrelated to the tile compression convention. Although this is an uncommon use case, it is permitted by the standard. (#159)
- Reworked some of the file I/O routines to allow simpler, more consistent mapping between OS-level file modes ('rb', 'wb', 'ab', etc.) and the more "PyFITS-specific" modes used by PyFITS like "readonly" and "update". That is, if reading a FITS file from an open file object, it doesn't matter as much what "mode" it was opened in so long as it has the right capabilities (read/write/etc.) Also works around bugs in the Python io module in 2.6+ with regard to file modes. (spacetelescope/PyFITS#33)
- Fixed an obscure issue that can occur on systems that don't have flush to memory-mapped files implemented (namely GNU Hurd). (astropy/astropy#968)

4.3.7 3.1.3 (2013-11-26)

- Disallowed assigning NaN and Inf floating point values as header values, since the FITS standard does not define a way to represent them in. Because this is undefined, the previous behavior did not make sense and produced invalid FITS files. (spacetelescope/PyFITS#11)
- Added a workaround for a bug in 64-bit OSX that could cause truncation when writing files greater than 2^32 bytes in size. (spacetelescope/PyFITS#28)

- Fixed a long-standing issue where writing binary tables did not correctly write the TFORMn keywords for variable-length array columns (they ommitted the max array length parameter of the format). This was thought fixed in v3.1.2, but it was only fixed there for compressed image HDUs and not for binary tables in general.
- Fixed an obscure issue that can occur on systems that don't have flush to memory-mapped files implemented (namely GNU Hurd). (Backported from 3.2)

4.3.8 3.0.12 (2013-11-26)

- Disallowed assigning NaN and Inf floating point values as header values, since the FITS standard does not define a way to represent them in. Because this is undefined, the previous behavior did not make sense and produced invalid FITS files. (Backported from 3.1.3)
- Added a workaround for a bug in 64-bit OSX that could cause truncation when writing files greater than 2^32 bytes in size. (Backported from 3.1.3)
- Fixed a long-standing issue where writing binary tables did not correctly write the TFORMn keywords for variable-length array columns (they ommitted the max array length parameter of the format). This was thought fixed in v3.1.2, but it was only fixed there for compressed image HDUs and not for binary tables in general. (Backported from 3.1.3)
- Fixed an obscure issue that can occur on systems that don't have flush to memory-mapped files implemented (namely GNU Hurd). (Backported from 3.2)

4.3.9 3.1.2 (2013-04-22)

- When an error occurs opening a file in fitsdiff the exception message will now at least mention which file had the error. (#168)
- Fixed support for opening gzipped FITS files by filename in a writeable mode (PyFITS has supported writing to gzip files for some time now, but only enabled it when GzipFile objects were passed to `pyfits.open()` due to some legacy code preventing full gzip support. (#195)
- Added a more helpful error message in the case of malformatted FITS files that contain non-float NULL values in an ASCII table but are missing the required TNULLn keywords in the header. (#197)
- Fixed an (apparently long-standing) issue where writing compressed images did not correctly write the TFORMn keywords for variable-length array columns (they ommitted the max array length parameter of the format). (#199)
- Slightly refactored how tables containing variable-length array columns are handled to add two improvements: Fixes an issue where accessing the data after a call to the `pyfits.getdata` convenience function caused an exception, and allows the VLA data to be read from an existing mmap of the FITS file. (#200)
- Fixed a bug that could occur when opening a table containing multi-dimensional columns (i.e. via the TDIMn keyword) and then writing it out to a new file. (#201)
- Added use of the console_scripts entry point to install the fitsdiff and fitscheck scripts, which if nothing else provides better Windows support. The generated scripts now override the ones explicitly defined in the scripts/ directory (which were just trivial stubs to begin with). (#202)
- Fixed a bug on Python 3 where attempting to open a non-existent file on Python 3 caused a seemingly unrelated traceback. (#203)
- Fixed a bug in fitsdiff that reported two header keywords containing NaN as value as different. (#204)
- Fixed an issue in the tests that caused some tests to fail if pyfits is installed with read-only permissions. (#208)

- Fixed a bug where instantiating a `BinTableHDU` from a numpy array containing boolean fields converted all the values to `False`. (#215)
- Fixed an issue where passing an array of integers into the constructor of `Column()` when the column type is floats of the same byte width caused the column array to become garbled. (#218)
- Fixed inconsistent behavior in creating CONTINUE cards from byte strings versus unicode strings in Python 2–CONTINUE cards can now be created properly from unicode strings (so long as they are convertable to ASCII). (spacetelescope/PyFITS#1)
- Fixed a couple cases where creating a new table using TDIMn in some of the columns could caused a crash. (spacetelescope/PyFITS#3)
- Fixed a bug in parsing HIERARCH keywords that do not have a space after the first equals sign (before the value). (spacetelescope/PyFITS#5)
- Prevented extra leading whitespace on HIERARCH keywords from being treated as part of the keyword. (spacetelescope/PyFITS#6)
- Fixed a bug where HIERARCH keywords containing lower-case letters was mistakenly marked as invalid during header validation. (spacetelescope/PyFITS#7)
- Fixed an issue that was ancillary to (spacetelescope/PyFITS#7) where the `Header.index()` method did not work correctly with HIERARCH keywords containing lower-case letters.

4.3.10 3.0.11 (2013-04-17)

- Fixed support for opening gzipped FITS files by filename in a writeable mode (PyFITS has supported writing to gzip files for some time now, but only enabled it when GzipFile objects were passed to `pyfits.open()` due to some legacy code preventing full gzip support. Backported from 3.1.2. (#195)
- Added a more helpful error message in the case of malformatted FITS files that contain non-float NULL values in an ASCII table but are missing the required TNULLn keywords in the header. Backported from 3.1.2. (#197)
- Fixed an (apparently long-standing) issue where writing compressed images did not correctly write the TFORMn keywords for variable-length array columns (they ommitted the max array length parameter of the format). Backported from 3.1.2. (#199)
- Slightly refactored how tables containing variable-length array columns are handled to add two improvements: Fixes an issue where accessing the data after a call to the `pyfits.getdata` convenience function caused an exception, and allows the VLA data to be read from an existing mmap of the FITS file. Backported from 3.1.2. (#200)
- Fixed a bug that could occur when opening a table containing multi-dimensional columns (i.e. via the TDIMn keyword) and then writing it out to a new file. Backported from 3.1.2. (#201)
- Fixed a bug on Python 3 where attempting to open a non-existent file on Python 3 caused a seemingly unrelated traceback. Backported from 3.1.2. (#203)
- Fixed a bug in fitsdiff that reported two header keywords containing NaN as value as different. Backported from 3.1.2. (#204)
- Fixed an issue in the tests that caused some tests to fail if pyfits is installed with read-only permissions. Backported from 3.1.2. (#208)
- Fixed a bug where instantiating a `BinTableHDU` from a numpy array containing boolean fields converted all the values to `False`. Backported from 3.1.2. (#215)
- Fixed an issue where passing an array of integers into the constructor of `Column()` when the column type is floats of the same byte width caused the column array to become garbled. Backported from 3.1.2. (#218)

- Fixed a couple cases where creating a new table using TDIMn in some of the columns could caused a crash. Backported from 3.1.2. (spacetelescope/PyFITS#3)

4.3.11 3.1.1 (2013-01-02)

This is a bug fix release for the 3.1.x series.

Bug Fixes

- Improved handling of scaled images and pseudo-unsigned integer images in compressed image HDUs. They now work more transparently like normal image HDUs with support for the `do_not_scale_image_data` and `uint` options, as well as `scale_back` and `save_backup`. The `.scale()` method works better too. (#88)
- Permits non-string values for the EXTNAME keyword when reading in a file, rather than throwing an exception due to the malformatting. Added verification for the format of the EXTNAME keyword when writing. (#96)
- Added support for EXTNAME and EXTVER in PRIMARY HDUs. That is, if EXTNAME is specified in the header, it will also be reflected in the `.name` attribute and in `pyfits.info()`. These keywords used to be verboten in PRIMARY HDUs, but the latest version of the FITS standard allows them. (#151)
- HCOMPRESS can again be used to compress data cubes (and higher-dimensional arrays) so long as the tile size is effectively 2-dimensional. In fact, PyFITS will automatically use compatible tile sizes even if they're not explicitly specified. (#171)
- Added support for the optional `endcard` parameter in the `Header.fromtextfile()` and `Header.totextfile()` methods. Although `endcard=False` was a reasonable default assumption, there are still text dumps of FITS headers that include the END card, so this should have been more flexible. (#176)
- Fixed a crash when running fitsdiff on two empty (that is, zero row) tables. (#178)
- Fixed an issue where opening files containing random groups HDUs in update mode could cause an unnecessary rewrite of the file even if none of the data is modified. (#179)
- Fixed a bug that could caused a deadlock in the filesystem on OSX if PyFITS is used with Numpy 1.7 in some cases. (#180)
- Fixed a crash when generating diff reports from diffs using the `ignore_comments` options. (#181)
- Fixed some bugs with 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)

4.3.12 3.0.10 (2013-01-02)

- Improved handling of scaled images and pseudo-unsigned integer images in compressed image HDUs. They now work more transparently like normal image HDUs with support for the `do_not_scale_image_data` and `uint` options, as well as `scale_back` and `save_backup`. The `.scale()` method works better too. Backported from 3.1.1. (#88)
- Permits non-string values for the EXTNAME keyword when reading in a file, rather than throwing an exception due to the malformatting. Added verification for the format of the EXTNAME keyword when writing. Backported from 3.1.1. (#96)
- Added support for EXTNAME and EXTVER in PRIMARY HDUs. That is, if EXTNAME is specified in the header, it will also be reflected in the `.name` attribute and in `pyfits.info()`. These keywords used to be verbotten in PRIMARY HDUs, but the latest version of the FITS standard allows them. Backported from 3.1.1. (#151)
- HCOMPRESS can again be used to compress data cubes (and higher-dimensional arrays) so long as the tile size is effectively 2-dimensional. In fact, PyFITS will not automatically use compatible tile sizes even if they're not explicitly specified. Backported from 3.1.1. (#171)
- Fixed a bug when writing out files containing zero-width table columns, where the TFIELDS keyword would be updated incorrectly, leaving the table largely unreadable. Backported from 3.1.0. (#174)
- Fixed an issue where opening files containing random groups HDUs in update mode could cause an unnecessary rewrite of the file even if none of the data is modified. Backported from 3.1.1. (#179)
- Fixed a bug that could caused a deadlock in the filesystem on OSX if PyFITS is used with Numpy 1.7 in some cases. Backported from 3.1.1. (#180)

4.3.13 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)

4.3.14 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)

4.3.15 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)

4.3.16 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)

4.3.17 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)

4.3.18 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)

4.3.19 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)

4.3.20 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.

4.3.21 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)

4.3.22 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)

4.3.23 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)

4.3.24 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)

4.3.25 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.

4.3.26 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.

4.3.27 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.

4.3.28 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)

4.3.29 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').

4.3.30 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.

4.3.31 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.

4.3.32 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)

4.3.33 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.

4.3.34 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.

4.3.35 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.

4.3.36 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.

4.3.37 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.

4.3.38 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.

4.3.39 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.

4.3.40 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.

4.3.41 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.

4.3.42 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

4.3.43 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.

4.3.44 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().

4.3.45 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.

4.3.46 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.

4.3.47 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.

4.3.48 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).