SPACE TELESCOPE SCIENCE INSTITUTE

PyFITS Documentation

Release 3.0.13.dev

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

January 27, 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 2.3 or newer. PyFITS also requires the numpy module. Information about numpy can be found at:

> http://numpy.scipy.org/

To download numpy, go to:

> http://sourceforge.net/project/numpy

PyFITS's source code is pure Python. It can be downloaded from:

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

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 Python's site-packages directory. If permissions do not allow this kind of installation PyFITS can be installed in a personal directory using one of the commands below. Note, that PYTHONPATH has to be set or modified accordingly. The three examples below show how to install PyFITS in an arbitrary directory <install-dir> and how to modify PYTHONPATH.

```
python setup.py install --home=<install-dir>
setenv PYTHONPATH <install-dir>/lib/python

python setup.py install --prefix=<install-lib>
setenv PYTHONPATH <install-dir>lib/python2.3/site-packages
```

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's 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 Python-like list, consisting of HDU objects. An HDU (Header Data Unit) is the highest level component of the FITS file structure. So, after the above open call, `hdulist[0]` is the primary HDU, `hdulist[1]`, if any, is the first extension HDU, etc. It should be noted that 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 cause the array data of each HDU to be accessed with mmap, rather than being read into memory all at once. This is particularly useful for working with very large arrays that cannot fit entirely into physical memory.

This has minimal impact on smaller files as well, though some operations, such as reading the array data sequentially, may incur some additional overhead. On 32-bit systems arrays larger than 2-3 GB cannot be mmap'd (which is fine, because by that point you're likely to run out of physical memory anyways), but 64-bit systems are much less limited in this respect.

Working With a FITS Header

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

The header attribute is a Header instance, another PyFITS object. To get the value of a header keyword, simply do (a la Python dictionaries):

```
>>> 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 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
```

Use the above syntax if the keyword is already present in the header. If the keyword might not exist and you want to add it if it doesn't, use the `Header.update()` method:

```
>>> prihdr.update('observer', 'Edwin Hubble')
```

Special methods must be used to add comment or history records:

```
>>> prihdr.add_history('I updated this file 2/26/09')
>>> prihdr.add_comment('Edwin Hubble really knew his stuff')
```

A header consists of `Card` objects (i.e. the 80-column card-images specified in the FITS standard). Each Card normally has up to three parts: key, value, and comment. To see the entire list of cardimages of an HDU, use the `Header.ascardlist()` method :

```
>>> print prihdr.ascardlist()[:3]
SIMPLE  =                    T / file does conform to FITS standard
BITPIX  =                   16 / number of bits per data pixel
NAXIS   =                    0 / number of data axes
```

Only the first three cards are shown above.

To get a list of all keywords, use the `CardList.keys()` method of the card list:

```
>>> prihdr.ascardlist().keys()
['SIMPLE', 'BITPIX', 'NAXIS', ...]
```

Working With Image Data

If an HDU's data is an image, the data attribute of the HDU object will return a numpy ndarray object. Refer to the numpy documentation for details on manipulating these numerical arrays.

```
>>> scidata = hdulist[1].data
```

Here, scidata points to the data object in the second HDU (the first HDU, `hdulist[0]`, being the primary HDU) in `hdulist`, which corresponds to the 'SCI' extension. Alternatively, you can access the extension by its extension name (specified in the EXTNAME keyword):

```
>>> scidata = hdulist['SCI'].data
```

If there is more than one extension with the same EXTNAME, EXTVER's value needs to be specified as the second argument, e.g.:

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

The returned numpy object has many attributes and methods for a user to get information about the array, e. g.:

```
>>> scidata.shape
(800, 800)
>>> scidata.dtype.name
'float32'
```

Since image data is a numpy object, we can slice it, view it, and perform mathematical operations on it. To see the pixel value at x=5, y=2:

```
>>> print scidata[1, 4]
```

Note that, like C (and unlike FORTRAN), Python is 0-indexed and the indices have the slowest axis first and fast axis last, i.e. for a 2-D image, the fast axis (X-axis) which corresponds to the FITS NAXIS1 keyword, is the second index. Similarly, the 1-indexed sub-section of x=11 to 20 (inclusive) and y=31 to 40 (inclusive) would be given in Python as:

```
>>> scidata[30:40, 10:20]
```

To update the value of a pixel or a sub-section:

```
>>> scidata[30:40, 10:20] = scidata[1, 4] = 999
```

This example changes the values of both the pixel [1, 4] and the sub-section [30:40, 10:20] to the new value of 999. See the Numpy documentation for more details on Python-style array indexing and slicing.

The next example of array manipulation is to convert the image data from counts to flux:

```
>>> photflam = hdulist[1].header['photflam']
>>> exptime = prihdr['exptime']
>>> scidata *= photflam / exptime
```

This example performs the math on the array in-place, thereby keeping the memory usage to a minimum.

If at this point you want to preserve all the changes you made and write it to a new file, you can use the `HDUList.writeto()` method (see below).

Working With Table Data

If you are familiar with the record array in numpy, you will find the table data is basically a record array with some extra properties. But familiarity with record arrays is not a prerequisite for this Guide.

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

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

```
>>> tbdata.field(0)
array([1, 2])
```

A numpy object with the data type of the specified field is returned.

Like header keywords, a field can be referred either by index, as above, or by name:

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

But how do we know what field names we've got? First, let's introduce another attribute of the table HDU: the `.columns` attribute:

```
>>> cols = hdulist[1].columns
```

This attribute is a `ColDefs` (column definitions) object. If we use the `ColDefs.info()` method:

```
>>> cols.info()
 name:
      ['c1', 'c2', 'c3', 'c4']
 format:
      ['1J', '3A', '1E', '1L']
 unit:
      ['', '', '', '']
 null:
      [-2147483647, '', '', '']
 bscale:
      ['', '', 3, '']
 bzero:
      ['', '', 0.40000000000000002, '']
 disp:
      ['I11', 'A3', 'G15.7', 'L6']
 start:
      ['', '', '', '']
 dim:
      ['', '', '', '']
```

it will show all its attributes, such as names, formats, bscales, bzeros, etc. We can also get these properties individually, e.g.:

```
>>> cols.names
['ID', 'name', 'mag', 'flag']
```

returns a (Python) list of field names.

Since each field is a numpy object, we'll have the entire arsenal of numpy tools to use. We can reassign (update) the values:

```
>>> tbdata.field('flag')[:] = 0
```

Save File Changes

As mentioned earlier, after a user opened a file, made a few changes to either header or data, the user can use `HDUList.writeto()` to save the changes. This takes the version of headers and data in memory and writes them to a new FITS file on disk. Subsequent operations can be performed to the data in memory and written out to yet another different file, all without recopying the original data to (more) memory.

```
>>> hdulist.writeto('newimage.fits')
```

will write the current content of `hdulist` to a new disk file newfile.fits. If a file was opened with the update mode, the `HDUList.flush()` method can also be used to write all the changes made since `open()`, back to the original file. The `close()` method will do the same for a FITS file opened with update mode.

```
>>> f = 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')
```

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:

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

As before, we create a `PrimaryHDU` object to encapsulate the data:

```
>>> hdu = pyfits.PrimaryHDU(n)
```

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

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

If this will be the only extension of the new FITS file and you only have a minimal primary HDU with no data, PyFITS again provides a short cut:

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

Alternatively, you can append it to the hdulist we have already created from the image file section:

```
>>> hdulist.append(tbhdu)
```

So far, we have covered the most basic features of 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.

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

```
>>> from 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
# the HDU with EXTNAME='sci' (if there is only 1)
>>> getheader('in.fits', 'sci')
# the HDU with EXTNAME='sci' and EXTVER=2
>>> getheader('in.fits', 'sci', 2)
>>> getheader('in.fits', ('sci', 2)) # use a tuple to do the same
>>> getheader('in.fits', ext=2) # the second extension
# the 'sci' extension, if there is only 1
>>> getheader('in.fits', extname='sci')
# the HDU with EXTNAME='sci' and EXTVER=2
>>> getheader('in.fits', extname='sci', extver=2)
# ambiguous specifications will raise an exception, DON'T DO IT!!
>>> getheader('in.fits', ext=('sci',1), extname='err', extver=2)
```

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

```
>>> from 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 keyword
                                          # FILTER's value
>>> val = getval('in.fits', 10, 'sci', 2) # get the 2nd sci extension's
                                           # 11th keyword's value
```

The function `getdata()` gets the data of an HDU. Similar to `getheader()`, it only requires the input FITS file name while the extension is specified through the optional arguments. It does have one extra optional argument header. If header is set to True, this function will return both data and header, otherwise only data is returned.

```
>>> from 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
```

1.3 FITS Headers

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

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, `.header` and `.data`.

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

```
>>> hdu = pyfits.PrimaryHDU()
>>> print hdu.header.ascardlist() # show the keywords
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 the required keywords from the input header first and then add back the required keywords compatible to the new HDU. So, a user can use a table HDU's header to construct an image HDU and vice versa. The constructor will also ensure the data type and dimension information in the header agree with the data.

1.3.2 The Header Attribute

Value Access and Updating

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 it's value
>>> print prihdr['darkcorr'] # get the value of the keyword 'darkcorr'
'OMIT'
>>> prihdr['darkcorr'] = 'PERFORM' # change darkcorr's value
```

When reference by the keyword name, it is case insensitive. Thus, prihdr['abc'], prihdr['ABC'], or prihdr['aBc'] are all equivalent.

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 2nd and 3rd keywords are removed from the original header.

Slices are not accepted by the header attribute, so it is not possible to do del `prihdr[3:5]`, for example.

The method `update(key, value, comment)` is a more versatile way to update keywords. It has the flexibility to update an existing keyword and in case the keyword does not exist, add it to the header. It also allows the use to update both the value and its comment. If it is a new keyword, the user can also specify where to put it, using the before or after optional argument. The default is to append at the end of the header.

```
>>> prihdr.update('target', 'NGC1234', 'target name')
>>> # place the next new keyword before the 'target' keyword
>>> prihdr.update('newkey', 666, before='target') # comment is optional
>>> # place the next new keyword after the 21st keyword
>>> prihdr.update('newkey2', 42.0, 'another new key', after=20)
```

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. Again, to get their values (except for the first one), a user must use indexing.

The following header methods are provided in PyFITS to add new commentary cards: `Header.add_history()`, `Header.add_comment()`, and `Header.add_blank()`. They are provided because the `Header.update()` method will not work - it will replace the first card of the same keyword.

Users can control where in the header to add the new commentary card(s) by using the optional before and after arguments, similar to the `update()` method used for regular cards. If no before or after is specified, the new card will be placed after the last one of the same kind (except blank-key cards which will always be placed at the end). If no card of the same kind exists, it will be placed at the end. Here is an example:

```
>>> hdu.header.add_history('history 1')
>>> hdu.header.add_blank('blank 1')
>>> hdu.header.add_comment('comment 1')
>>> hdu.header.add_history('history 2')
>>> hdu.header.add_blank('blank 2')
>>> hdu.header.add_comment('comment 2'))
```

and the part in the modified header becomes:

```
HISTORY history 1
HISTORY history 2
        blank 1
COMMENT comment 1
COMMENT comment 2
        blank 2
```

Ironically, there is no comment in a commentary card , only a string value.

1.3.3 Card Images

A FITS header consists of card images.

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

Cards have the attributes `.key`, `.value`, and `.comment`. Both `.value` and `.comment` can be changed but not the `.key` attribute.

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

1.3.4 Card List

The Header itself only has limited functionality. Many lower level operations can only be achieved by going through its `CardList` object.

The header is basically a list of `Card` objects. This list can be manifested as a `CardList` object in PyFITS. It is accessed via the `Header.ascardlist()` method (or the `.ascard` attribute, for short). Since the header attribute only refers to a card value, so when a user needs to access a card's other properties (e.g. the comment) in a header, it has to go through the `CardList`.

Like the header's item, the `CardList`'s item can be accessed through either the keyword name or index.

```
>>> cards = prihdr.header.ascardlist()
>>> cards['abc'].comment = 'new comment' # update the keyword ABC's comment
>>> cards[3].key # see the keyword name of the 4th card
>>> cards[10:20].keys() # see keyword names from cards 11 to 20
```

1.3.5 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.

```
>>> c = pyfits.Card('abc', 'abcdefg'*20)
>>> print c
ABC = 'abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcd&'
CONTINUE 'efgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefga&'
CONTINUE 'bcdefg&'
>>> c.value
'abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgab
cdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefg'
# both value and comments are long
>>> c = pyfits.Card('abc', 'abcdefg'*10, 'abcdefg'*10)
>>> print c
ABC = 'abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcd&'
CONTINUE 'efg&'
CONTINUE '&' / abcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefgabcdefga
CONTINUE '&' / bcdefg
```

Note that when 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 considered a Card. So, if it is the nth card in a header, the (n+1)th card refers to the next keyword, not the 80-characters containing CONTINUE. These keywords having long string values can be accessed and updated just like regular keywords.

1.3.6 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.

When creating or updating using the `Header.update()` method, it is necessary to prepend 'hierarch' (case insensitive). But if the keyword is already in the header, it can be accessed or updated by assignment by using the keyword name diretly, with or without the 'hierarch' prepending. The keyword name will preserve its cases from its constructor, but when referring to the keyword, it is case insensitive.

Examples follow:

```
>>> c = pyfits.Card('abcdefghi', 10)
...
ValueError: keyword name abcdefghi is too long (> 8), use HIERARCH.
>>> c = pyfits.Card('hierarch abcdefghi', 10)
>>> print c
HIERARCH abcdefghi = 10
>>> h = pyfits.PrimaryHDU()
>>> h.header.update('hierarch abcdefghi', 99)
>>> h.header.update('hierarch abcdefghi', 99)
>>> h.header['abcdefghi']
99
>>> h.header['abcdefghi'] = 10
>>> h.header['hierarch abcdefghi']
10
# case insensitive
>>> h.header.update('hierarch ABCdefghi', 1000)
>>> print h.header
SIMPLE = T / conforms to FITS standard
BITPIX = 8 / array data type
NAXIS = 0 / number of array dimensions
EXTEND = T
HIERARCH ABCdefghi = 1000
>>> h.header['hierarch abcdefghi']
1000
```

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 as specified by the BITPIX keyword. For scaled image, the .data attribute will be the physical data, i.e. already transformed from the storage data and may not be the same data type as prescribed in BITPIX. This means an extra step of copying is needed and thus the corresponding memory requirement. This also means that the advantage of memory mapping is reduced for scaled data.

For floating point storage data, the scaled data will have the same data type. For integer data type, the scaled data will always be single precision floating point (numpy.float32). Here is an example of what happens to such a file, before and after the data is touched

```
>>> f = 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).

Writing Scaled Image Data

With the extra processing and memory requirement, we discourage users to use scaled data as much as possible. However, PyFITS does provide ways to write scaled data with the scale(type, option, bscale, bzero) method. Here are a few examples:

```
>>> # scale the data to Int16 with user specified bscale/bzero
>>> hdu.scale('int16', '', bzero=32768)
>>> # scale the data to Int32 with the min/max of the data range
>>> hdu.scale('int32', 'minmax')
>>> # scale the data, using the original BSCALE/BZERO
>>> hdu.scale('int32', 'old')
```

The first example above shows how to store an unsigned short integer array.

Great caution must be exercised when using the `scale()` method. The `.data` attribute of an image HDU, after the `scale()` call, will become the storage values, not the physical values. So, only call `scale()` just before writing out to FITS files, i.e. calls of `writeto()`, `flush()`, or `close()`. No further use of the data should be exercised. Here is an example of what happens to the `.data` attribute after the `scale()` call:

```
>>> hdu = 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 HDU's being accessed at the same time, the system may run out of memory.

If a user does not need the entire image(s) at the same time, e.g. processing images(s) ten rows at a time, the `section` attribute of an HDU can be used to alleviate such memory problems.

With PyFITS' improved support for memory-mapping, the sections feature is not as necessary as it used to be for handling very large images. However, if the image's data is scaled with non-trivial BSCALE/BZERO values, accessing the data in sections may still be necessary under the current implementation. Memmap is also insufficient for loading images large than ~4 GB on a 32-bit system–in such cases it may be necessary to use sections.

Here is an example of getting the median image from 3 input images of the size 5000x5000:

```
>>> f1 = 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 memory.

Sections cannot 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 takes up more storage space as well as more processing time since the ASCII text need to be parsed back 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.

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:

```
>>> print tbhdu.header.ascardlist()
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. All other objects (e.g. `CardList`) do not have any `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:

```
>>> print 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).
>>> print 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:

```
>>> print hdu.header.ascardlist()[4:] # has a bunch of fixable cards
FIX1 = 2.1e23
FIX2= 2
FIX3 = string value without quotes
FIX4 2
FIX5 = 2.4 e 03
FIX6 = '2 10 '
# can still access the values before the fix
>>> hdu.header[5]
2
>>> hdu.header['fix4']
2
>>> hdu.header['fix5']
2400.0
>>> hdu.verify('silentfix')
>>> print hdu.header.ascard[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.update('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'")
# then append it to the header (must go through the CardList)
>>> h.header.ascardlist().append(c)
# Now if we try to write to a FITS file, the default output verification
# will not take it.
>>> h.writeto('pi.fits')
Output verification result:
HDU 0:
  Card 4:
    Unfixable error: Illegal keyword name 'P.I.'
......
  raise VerifyError
VerifyError
# Must set the output_verify argument to 'ignore', to force writing a
# non-standard FITS file
>>> h.writeto('pi.fits', output_verify='ignore')
# Now reading a non-standard FITS file
# pyfits is magnanimous in reading non-standard FITS file
>>> hdus = pyfits.open('pi.fits')
>>> print hdus[0].header.ascardlist()
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.
>>> print hdu[1].header.ascardlist()
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 `.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')
>>> print hdu.header.ascardlist()[:]
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 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
"""
```

1.8 Miscellaneous Features

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

1.8.1 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 Reference Manual

Examples

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

```
#!/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()
```

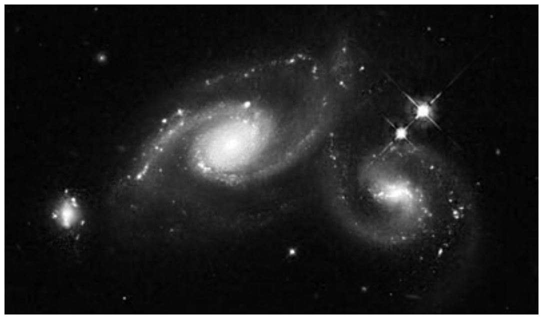


Figure 1.1: Red color information

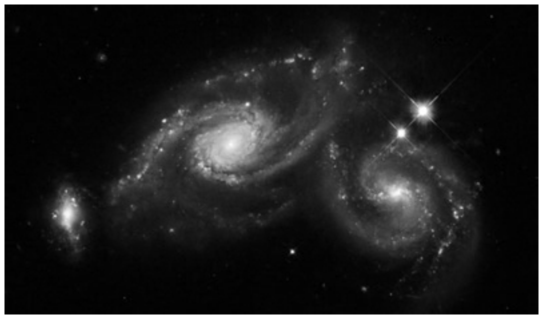


Figure 1.2: Green color information

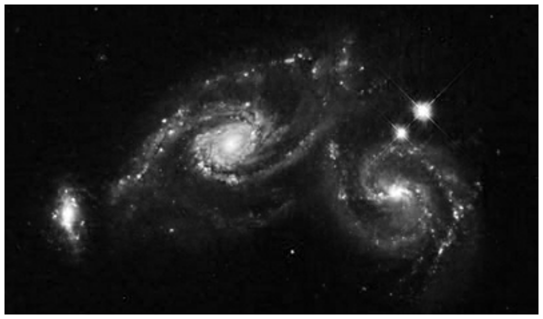


Figure 1.3: Blue color information

```
xsize, ysize = image.size
r, g, b = image.split()
rdata = r.getdata() # data is now an array of length ysize\*xsize
gdata = g.getdata()
bdata = b.getdata()

# create numpy arrays
npr = numpy.reshape(rdata, (ysize, xsize))
npg = numpy.reshape(gdata, (ysize, xsize))
npb = numpy.reshape(bdata, (ysize, xsize))

# write out the fits images, the data numbers are still JUST the RGB
# scalings; don't use for science
red = pyfits.PrimaryHDU()
red.header.update('LATOBS', "32:11:56") # add spurious header info
red.header.update('LONGOBS', "110:56")
red.data = npr
red.writeto('red.fits')
green = pyfits.PrimaryHDU()
green.header.update('LATOBS', "32:11:56")
green.header.update('LONGOBS', "110:56")
green.data = npg
green.writeto('green.fits')
blue = pyfits.PrimaryHDU()
blue.header.update('LATOBS', "32:11:56")
blue.header.update('LONGOBS', "110:56")
blue.data = npb
blue.writeto('blue.fits')
```

CHAPTER 2

API Documentation

2.1 Opening Files

`pyfits.open(*args, **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?

classExtensions : dict (''Deprecated'')

A dictionary that maps pyfits classes to extensions of those classes. When present in the dictionary, the extension class will be constructed in place of the pyfits class.

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

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 NOT be preserved unless the file was opened with `checksum=True`. This behavior may change in a future PyFITS version.

- **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.

Returns

hdulist : an `HDUList` object

`HDUList` containing all of the header data units in the file.

2.1.1 Convenience functions

The functions in this module provide shortcuts for some of the most basic operations on FITS files, such as reading and updating the header. They are included directly in the 'pyfits' namespace so that they can be used like:

```
>>> pyfits.getheader(...)
```

These functions are primarily for convenience when working with FITS files in the command-line interpreter. If performing several operations on the same file, such as in a script, it is better to *not* use these functions, as each one must open and re-parse the file. In such cases it is better to use `pyfits.open()` and work directly with the `pyfits.HDUList` object and underlying HDU objects.

Several of the convenience functions, such as `getheader` and `getdata` support special arguments for selecting which extension HDU to use when working with a multi-extension FITS file. There are a few supported argument formats for selecting the extension. See the documentation for `getdata` for an explanation of all the different formats.

Warning: All arguments to convenience functions other than the filename that are *not* for selecting the extension HDU should be passed in as keyword arguments. This is to avoid ambiguity and conflicts with the extension arguments. For example, to set NAXIS=1 on the Primary HDU:
Wrong:

```
>>> pyfits.setval('myimage.fits', 'NAXIS', 1)
```

The above example will try to set the NAXIS value on the first extension HDU to blank. That is, the argument '1' is assumed to specify an extension HDU.
Right:

```
>>> pyfits.setval('myimage.fits', 'NAXIS', value=1)
```

This will set the NAXIS keyword to 1 on the primary HDU (the default). To specify the first extension HDU use:

```
>>> pyfits.setval('myimage.fits', 'NAXIS', value=1, ext=1)
```

This complexity arises out of the attempt to simultaneously support multiple argument formats that were used in past versions of PyFITS. Unfortunately, it is not possible to support all formats without introducing some ambiguity. A future PyFITS release may standardize around a single format and offically deprecate the other formats.

`pyfits.convenience.`**`getdata`**(*filename*, **args*, ***kwargs*)

Get the data from an extension of a FITS file (and optionally the header).

Parameters

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

File to get data from. If opened, mode must be one of the following rb, rb+, or ab+.

ext :

The rest of the arguments are for extension specification. They are flexible and are best illustrated by examples.

No extra arguments implies the primary header:

```
>>> getdata('in.fits')
```

By extension number:

```
>>> getdata('in.fits', 0)     # the primary header
>>> getdata('in.fits', 2)     # the second extension
>>> getdata('in.fits', ext=2) # the second extension
```

By name, i.e., `EXTNAME` value (if unique):

```
>>> getdata('in.fits', 'sci')
>>> getdata('in.fits', extname='sci') # equivalent
```

Note `EXTNAME` values are not case sensitive

By combination of `EXTNAME` and EXTVER`` as separate arguments or as a tuple:

```
>>> getdata('in.fits', 'sci', 2) # EXTNAME='SCI' & EXTVER=2
>>> getdata('in.fits', extname='sci', extver=2) # equivalent
>>> getdata('in.fits', ('sci', 2)) # equivalent
```

Ambiguous or conflicting specifications will raise an exception:

```
>>> getdata('in.fits', ext=('sci',1), extname='err', extver=2)
```

header : bool (optional)

If `True`, return the data and the header of the specified HDU as a tuple.

lower, upper : bool (optional)

If `lower` or `upper` are `True`, the field names in the returned data object will be converted to lower or upper case, respectively.

view : ndarray (optional)

When given, the data will be turned wrapped in the given ndarray subclass by calling:

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

kwargs :

Any additional keyword arguments to be passed to `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.

`pyfits.convenience.`**`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

`pyfits.convenience.`**`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

`pyfits.convenience.`**`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.

`pyfits.convenience.`**`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.

`pyfits.convenience.`**`writeto`**(*filename*, *data*, *header=None*, *output_verify='exception'*, *clobber=False*, *checksum=False*)

Create a new FITS file using the supplied data/header.

Parameters

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

File to write to. If opened, must be opened for append (ab+).

data : array, record array, or groups data object

data to write to the new file

header : `Header` object (optional)

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

output_verify : str

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

clobber : bool (optional)

If `True`, and if filename already exists, it will overwrite the file. Default is `False`.

checksum : bool (optional)

If `True`, adds both `DATASUM` and `CHECKSUM` cards to the headers of all HDU's written to the file.

`pyfits.convenience.`**`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`.

`pyfits.convenience.`**`update`**(*filename*, *data*, **args*, ***kwargs*)

Update the specified extension with the input data/header.

Parameters

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

File to update. If opened, mode must be update (rb+). An opened file object or `GzipFile` object will be closed upon return.

data : array, table, or group data object

the new data used for updating

header : `Header` object (optional)

The header associated with `data`. If `None`, an appropriate header will be created for the data object supplied.

ext, extname, extver :

The rest of the arguments are flexible: the 3rd argument can be the header associated with the data. If the 3rd argument is not a `Header`, it (and other positional arguments) are assumed to be the extension specification(s). Header and extension specs can also be keyword arguments. For example:

```
>>> update(file, dat, hdr, 'sci')  # update the 'sci' extension
>>> update(file, dat, 3)  # update the 3rd extension
>>> update(file, dat, hdr, 3)  # update the 3rd extension
>>> update(file, dat, 'sci', 2)  # update the 2nd SCI extension
>>> update(file, dat, 3, header=hdr)  # update the 3rd extension
>>> update(file, dat, header=hdr, ext=5) # update the 5th extension
```

kwargs :

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

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

Print the summary information on a FITS file.

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

Parameters

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

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

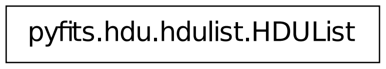
output : file, bool (optional)

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.

2.2 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 `_BaseHDU`.

file : file object, optional

The opened physical file associated with the `HDUList`.

`append``(`**args*, ***kwargs*`)`

Append a new HDU to the `HDUList`.

Parameters

hdu : instance of _BaseHDU

HDU to add to the `HDUList`.

classExtensions : dict

A dictionary that maps pyfits classes to extensions of those classes. When present in the dictionary, the extension class will be constructed in place of the pyfits class.

`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

dictionary 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

classExtensions : dict

A dictionary that maps pyfits classes to extensions of those classes. When present in the dictionary, the extension class will be constructed in place of the pyfits class.

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`(**args*, ***kwargs*)

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

Parameters

index : int

Index before which to insert the new HDU.

hdu : _BaseHDU instance

The HDU object to insert

classExtensions : dict

A dictionary that maps pyfits classes to extensions of those classes. When present in the dictionary, the extension class will be constructed in place of the pyfits class.

`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.

`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.

`writeto`(**args*, ***kwargs*)

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.

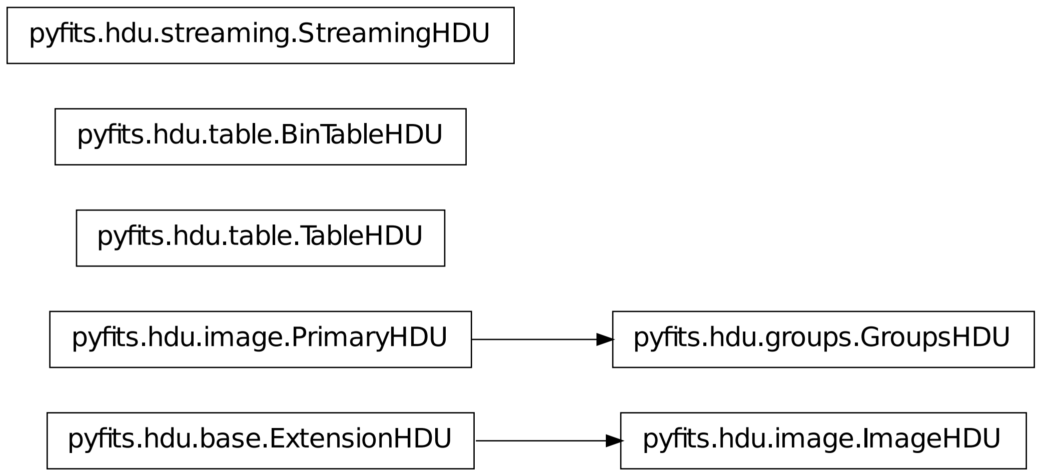
clobber : bool

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


pyfits.hdu.streaming.StreamingHDU
pyfits.hdu.table.BinTableHDU
pyfits.hdu.table.TableHDU
pyfits.hdu.image.PrimaryHDU
pyfits.hdu.groups.GroupsHDU
pyfits.hdu.base.ExtensionHDU
pyfits.hdu.image.ImageHDU

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*)

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.

`add_checksum``(`*when=None*, *override_datasum=False*, *blocking='standard'*`)`

Add the `CHECKSUM` and `DATASUM` cards to this HDU with the values set to the checksum calculated for the HDU and the data respectively. The addition of the `DATASUM` card may be overridden.

Parameters

when : str, optional

comment string for the cards; by default the comments will represent the time when the checksum was calculated

override_datasum : bool, optional

add the `CHECKSUM` card only

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Notes

For testing purposes, first call `add_datasum` with a `when` argument, then call `add_checksum` with a `when` argument and `override_datasum` set to `True`. This will provide consistent comments for both cards and enable the generation of a `CHECKSUM` card with a consistent value.

`add_datasum``(`*when=None*, *blocking='standard'*`)`

Add the `DATASUM` card to this HDU with the value set to the checksum calculated for the data.

Parameters

when : str, optional

Comment string for the card that by default represents the time when the checksum was calculated

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns

checksum : int

The calculated datasum

Notes

For testing purposes, provide a `when` argument to enable the comment value in the card to remain consistent. This will enable the generation of a `CHECKSUM` card with a consistent value.

`copy``()`

Make a copy of the HDU, both header and data are copied.

`filebytes``()`

Calculates and returns the number of bytes that this HDU will write to a file.

Parameters

None :

Returns

Number of bytes :

fileinfo()

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

Parameters

None :

Returns

dictionary or None :

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

Dictionary contents:

Key	Value
file	File object associated with the HDU
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.

Parameters

data : str

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 match_header(*header*)

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

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

Parameters

fileobj : file object or file-like object

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

checksum : bool

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

ignore_missing_end : bool

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

classmethod `register_hdu`(*hducls*)

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

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

TODO: Write about parameters

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

`run_option`(*option='warn'*, *err_text=''*, *fix_text='Fixed.'*, *fix=None*, *fixable=True*)

Execute the verification with selected option.

`scale`(*type=None*, *option='old'*, *bscale=1*, *bzero=0*)

Scale image data by using `BSCALE`/`BZERO`.

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

Parameters

type : str, optional

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

option : str

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

bscale, bzero : int, optional

User-specified `BSCALE` and `BZERO` values.

`size`()

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

classmethod `unregister_hdu`(*hducls*)

`update_ext_name`(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)

Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

value to be used for the new extension name

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

`update_ext_version`(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)

Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

value to be used for the new extension version

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

`update_header`()

`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`(**args*, ***kwargs*)
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.

classExtensions : dict

A dictionary that maps pyfits classes to extensions of those classes. When present in the dictionary, the extension class will be constructed in place of the pyfits class.

checksum : bool

When `True` adds both `DATASUM` and `CHECKSUM` cards to the header of the HDU when written to the file.

`ImgCode` **= {'uint64': 64, 'uint16': 16, 'int16': 16, 'int64': 64, 'int32': 32, 'float64': -64, 'uint8': 8, 'float32': -32, 'uint32'**

`NumCode` **= {-64: 'float64', -32: 'float32', 32: 'int32', 8: 'uint8', 64: 'int64', 16: 'int16'}**

`data`

Works similarly to property(), but computes the value only once.

Adapted from the recipe at http://code.activestate.com/recipes/363602-lazy-property-evaluation

`header`

`is_image`

`name`

`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`.

`standard_keyword_comments` **= {'BITPIX': 'array data type', 'XTENSION': 'Image extension', 'SIMPLE': 'confor**

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.

TODO: Write me

`add_checksum`(*when=None*, *override_datasum=False*, *blocking='standard'*)

Add the `CHECKSUM` and `DATASUM` cards to this HDU with the values set to the checksum calculated for the HDU and the data respectively. The addition of the `DATASUM` card may be overridden.

Parameters

when : str, optional

comment string for the cards; by default the comments will represent the time when the checksum was calculated

override_datasum : bool, optional

add the `CHECKSUM` card only

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Notes

For testing purposes, first call `add_datasum` with a `when` argument, then call `add_checksum` with a `when` argument and `override_datasum` set to `True`. This will provide consistent comments for both cards and enable the generation of a `CHECKSUM` card with a consistent value.

add_datasum(*when=None*, *blocking='standard'*)

Add the `DATASUM` card to this HDU with the value set to the checksum calculated for the data.

Parameters

when : str, optional

Comment string for the card that by default represents the time when the checksum was calculated

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns

checksum : int

The calculated datasum

Notes

For testing purposes, provide a `when` argument to enable the comment value in the card to remain consistent. This will enable the generation of a `CHECKSUM` card with a consistent value.

copy()

Make a copy of the HDU, both header and data are copied.

filebytes()

Calculates and returns the number of bytes that this HDU will write to a file.

Parameters

None :

Returns

Number of bytes :

fileinfo()

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

Parameters

None :

Returns

dictionary or None :

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

Dictionary contents:

Key	Value
file	File object associated with the HDU
file-mode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
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.

Parameters

data : str

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 `match_header`(*header*)

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

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

Parameters

fileobj : file object or file-like object

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

checksum : bool

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

ignore_missing_end : bool

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

classmethod `register_hdu`(*hducls*)

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

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

TODO: Write about parameters

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

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

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

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

Parameters
type : str, optional

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

option : str

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

bscale, bzero : int, optional

User-specified `BSCALE` and `BZERO` values.

size()
Returns the size (in bytes) of the HDU's data part.

classmethod unregister_hdu(*hducls*)

update_ext_name(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)
Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters
value : str

value to be used for the new extension name

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

update_ext_version(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)
Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

value to be used for the new extension version

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

update_header()

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`(**args*, ***kwargs*)
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.

classExtensions : dict

A dictionary that maps pyfits classes to extensions of those classes. When present in the dictionary, the extension class will be constructed in place of the pyfits class.

checksum : bool

When `True` adds both `DATASUM` and `CHECKSUM` cards to the header of the HDU when written to the file.

`ImgCode` = {'uint64': 64, 'uint16': 16, 'int16': 16, 'int64': 64, 'int32': 32, 'float64': -64, 'uint8': 8, 'float32': -32, 'uint32'

`NumCode` = {-64: 'float64', -32: 'float32', 32: 'int32', 8: 'uint8', 64: 'int64', 16: 'int16'}

`columns`
Works similarly to property(), but computes the value only once.

Adapted from the recipe at http://code.activestate.com/recipes/363602-lazy-property-evaluation

`data`
Works similarly to property(), but computes the value only once.

Adapted from the recipe at http://code.activestate.com/recipes/363602-lazy-property-evaluation

`header`

`is_image`

`name`

parnames
Works similarly to property(), but computes the value only once.

Adapted from the recipe at http://code.activestate.com/recipes/363602-lazy-property-evaluation

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`.

standard_keyword_comments = {'BITPIX': 'array data type', 'XTENSION': 'Image extension', 'SIMPLE': 'confor

2.3.3 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.update(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 for append (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.

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

write(*data*)
Write the given data to the stream.

Parameters
data : ndarray

Data to stream to the file.

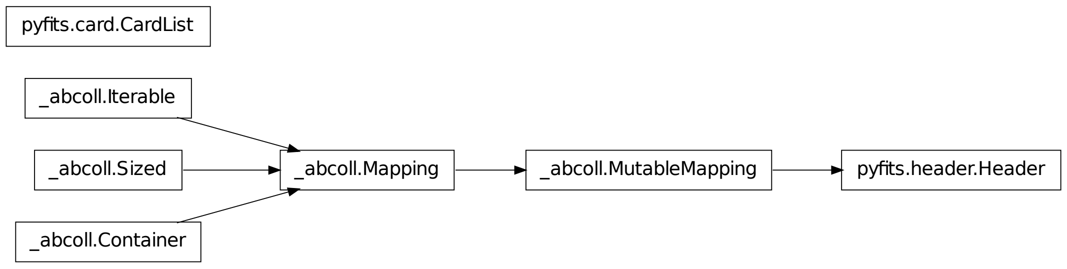
Returns
writecomplete : int

Flag that when `True` indicates that all of the required data has been written to the stream.

Notes

Only the amount of data specified in the header provided to the class constructor may be written to the stream. If the provided data would cause the stream to overflow, an `IOError` exception is raised and the data is not written. Once sufficient data has been written to the stream to satisfy the amount specified in the header, the stream is padded to fill a complete FITS block and no more data will be accepted. An attempt to write more data after the stream has been filled will raise an `IOError` exception. If the dtype of the input data does not match what is expected by the header, a `TypeError` exception is raised.

2.4 Headers



2.4.1 `Header`

class `pyfits.`**Header**(*cards=*[ ], *txtfile=None*)
Bases: `_abcoll.MutableMapping`

FITS header class.

The purpose of this class is to present the header like a dictionary as opposed to a list of cards.

The attribute `ascard` supplies the header like a list of cards.

The header class uses the card's keyword as the dictionary key and the cards value is the dictionary value.

The `has_key`, `get`, and `keys` methods are implemented to provide the corresponding dictionary functionality. The header may be indexed by keyword value and like a dictionary, 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.

The header may also be indexed by card list index number. In that case, the value of the card at the given index in the card list will be returned.

A delete method has been implemented to allow deletion from the header. When `del` is called, all cards with the given keyword are deleted from the header.

The `Header` class has an associated iterator class `_Header_iter` which will allow iteration over the unique keywords in the header dictionary.

Construct a `Header` from a `CardList` 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.

`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`

`ascardlist`(**args*, ***kwargs*)

Returns a `CardList` object.

`clear`()

`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.

`fromTxtFile`(*fileobj*, *replace=False*)

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 `fromstring`(*data*)

Creates an HDU header from a byte string containing the entire header data.

Parameters

data : str

String containing the entire header.

`get`(*key*, *default=None*)

`get_comment`()

Get all comment cards as a list of string texts.

`get_history`()

Get all history cards as a list of string texts.

`has_key`(**args*, ***kwargs*)

Check for existence of a keyword.

Parameters

key : str or int

Keyword name. If given an index, always returns 0.

Returns

has_key : bool

Returns `True` if found, otherwise, `False`.

items()
Override items since the default implementation does not propertly handle duplicate keywords.

iteritems()
Override iteritems since the default implementation does not properly handle duplicate keywords.

iterkeys()

itervalues()
Override itervalues since the default implementation does not properly handle duplicate keywords.

keys()
Return a list of keys with duplicates removed.

> **Warning:** There is a surprising incogruity in Header objecets between `Header.keys()` and `Header.iterkeys()`. The latter does *not* remove duplicates. This incongruity exists for historical reasons, but is not be design. In PyFITS 3.1 it is done away with, and :meth:Header.keys returns the exact keywords appearin the header, including duplicates.

pop(*key*, *default=<object object at 0x7fab9918b030>*)

popitem()

rename_key(*oldkey*, *newkey*, *force=False*)
Rename a card's keyword in the header.

Parameters

oldkey : str or int

old keyword

newkey : str

new keyword

force : bool

When `True`, if new key name already exists, force to have duplicate name.

setdefault(*keyword*, *default=''*)
PyFITS < 3.1 won't allow item assignment to keywords that don't already exist, but for the setdefault dict method to work at all, it needs to be able to add nonexistent keywords with the default value.

toTxtFile(*fileobj*, *clobber=False*)
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.

update(*key*, *value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)
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 or 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 or 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``()`

Override values since the default implementation does not properly handle duplicate keywords.

2.4.2 `CardList`

class `pyfits.`**`CardList`**`(`*cards=*[ ], *keylist=None*`)`

Bases: `list`

FITS header card list class.

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

Parameters

cards :

A list of `Card` objects.

`append``(`*card*, *useblanks=True*, *bottom=False*`)`

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`()
Make a (deep)copy of the `CardList`.

`count_blanks`()
Returns how many blank cards are *directly* before the `END` card.

`filterList`(*key*)
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.

`filter_list`(*key*)
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_of`(*key*, *backward=False*)
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`(*pos*, *card*, *useblanks=True*)
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`()

Return a list of all keywords from the `CardList`.

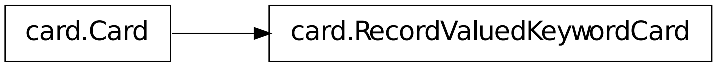
Keywords include `field_specifier` for `RecordValuedKeywordCard` objects.

`values`()

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`.

2.5 Cards



2.5.1 `Card`

class `pyfits.`**`Card`**(*key=''*, *value=''*, *comment=''*)

Bases: `pyfits.verify._Verify`

Construct a card from `key`, `value`, and (optionally) `comment`. Any specifed arguments, except defaults, must be compliant to FITS standard.

Parameters

key : str, optional

keyword name

value : str, optional

keyword value

comment : str, optional

comment

`ascardimage`(**args*, ***kwargs*)

classmethod `fromstring`(*cardimage*)

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).

`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`

`comment`

Card comment

`key`

Card keyword

`length` = 80

`value`

Card value

2.5.2 `RecordValuedKeywordCard`

class `pyfits.`**`RecordValuedKeywordCard`**(*key=''*, *value=''*, *comment=''*)

Bases: `pyfits.card.Card`

Class to manage record-valued keyword cards as described 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 grammar for field-specifier is:

```
field-specifier:
    field
    field-specifier.field

field:
    identifier
    identifier.index
```

where `identifier` is a sequence of letters (upper or lower case), underscores, and digits of which the first character must not be a digit, and `index` is a sequence of digits. No blank characters may occur in the field-specifier. The `index` is provided primarily for defining array elements though it need not be used for that purpose.

Multiple record-valued keywords of the same name but differing values may be present in a FITS header. The field-specifier may be viewed as part of the keyword name.

Some examples follow:

```
DP1     = 'NAXIS: 2'
DP1     = 'AXIS.1: 1'
DP1     = 'AXIS.2: 2'
DP1     = 'NAUX: 2'
DP1     = 'AUX.1.COEFF.0: 0'
DP1     = 'AUX.1.POWER.0: 1'
DP1     = 'AUX.1.COEFF.1: 0.00048828125'
DP1     = 'AUX.1.POWER.1: 1'
```

Parameters

key : str, optional

The key, either the simple key or one that contains a field-specifier

value : str, optional

The value, either a simple value or one that contains a field-specifier

comment : str, optional

The comment

ascardimage(**args*, ***kwargs*)

classmethod create(*key=''*, *value=''*, *comment=''*)

classmethod coerce(*card*)

Coerces an input `Card` object to a `RecordValuedKeywordCard` object if the value of the card meets the requirements of this type of card.

Parameters

card : `Card` object

A `Card` object to coerce

Returns

card :

•If the input card is coercible:

a new `RecordValuedKeywordCard` constructed from the `key`, `value`, and `comment` of the input card.

•If the input card is not coercible:

the input card

classmethod create(*key='*', *value='*', *comment='*')

Create a card given the input `key`, `value`, and `comment`. If the input key and value qualify for a `RecordValuedKeywordCard` then that is the object created. Otherwise, a standard `Card` object is created.

Parameters

key : str, optional

The key

value : str, optional

The value

comment : str, optional

The comment

Returns

card :

Either a `RecordValuedKeywordCard` or a `Card` object.

classmethod `createCard`(**args*, ***kwargs*)

Create a card given the input `key`, `value`, and `comment`. If the input key and value qualify for a `RecordValuedKeywordCard` then that is the object created. Otherwise, a standard `Card` object is created.

Parameters

key : str, optional

The key

value : str, optional

The value

comment : str, optional

The comment

Returns

card :

Either a `RecordValuedKeywordCard` or a `Card` object.

classmethod `createCardFromString`(**args*, ***kwargs*)

Create a card given the `input` string. If the `input` string can be parsed into a key and value that qualify for a `RecordValuedKeywordCard` then that is the object created. Otherwise, a standard `Card` object is created.

Parameters

input : str

The string representing the card

Returns

card :

either a `RecordValuedKeywordCard` or a `Card` object

classmethod `fromstring`(*input*)

Create a card given the `input` string. If the `input` string can be parsed into a key and value that qualify for a `RecordValuedKeywordCard` then that is the object created. Otherwise, a standard `Card` object is created.

Parameters

input : str

The string representing the card

Returns

card :

either a `RecordValuedKeywordCard` or a `Card` object

`run_option`(*option='warn'*, *err_text=''*, *fix_text='Fixed.'*, *fix=None*, *fixable=True*)

Execute the verification with selected option.

strvalue()

Method to extract the field specifier and value from the card image. This is what is reported to the user when requesting the value of the `Card` using either an integer index or the card key without any field specifier.

classmethod upperKey(**args*, ***kwargs*)

`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 : int or str

A keyword value that could be an integer, a key, or a `key.field-specifier` value

Returns

Integer input :

the original integer key

String input :

the converted string

classmethod upper_key(*key*)

`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 : int or str

A keyword value that could be an integer, a key, or a `key.field-specifier` value

Returns

Integer input :

the original integer key

String input :

the converted string

classmethod validKeyValue(**args*, ***kwargs*)

Determine if the input key and value can be used to form a valid `RecordValuedKeywordCard` object. The `key` parameter may contain the key only or both the key and field-specifier. The `value` may be the value only or the field-specifier and the value together. The `value` parameter is optional, in which case the `key` parameter must contain both the key and the field specifier.

Parameters

key : str

The key to parse

value : str or float-like, optional

The value to parse

Returns

valid input : A list containing the key, field-specifier, value

invalid input : An empty list

Examples

```
>>> valid_key_value('DP1','AXIS.1: 2')
>>> valid_key_value('DP1.AXIS.1', 2)
>>> valid_key_value('DP1.AXIS.1')
```

classmethod `valid_key_value` (*key*, *value=0*)

Determine if the input key and value can be used to form a valid `RecordValuedKeywordCard` object. The `key` parameter may contain the key only or both the key and field-specifier. The `value` may be the value only or the field-specifier and the value together. The `value` parameter is optional, in which case the `key` parameter must contain both the key and the field specifier.

Parameters

key : str

The key to parse

value : str or float-like, optional

The value to parse

Returns

valid input : A list containing the key, field-specifier, value

invalid input : An empty list

Examples

```
>>> valid_key_value('DP1','AXIS.1: 2')
>>> valid_key_value('DP1.AXIS.1', 2)
>>> valid_key_value('DP1.AXIS.1')
```

`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`

`comment`

Card comment

`field` **= ‘[a-zA-Z_]\\w*(\\.\\d+)?’**

`field_specifier`

`field_specifier_NFSC_image_RE` **= <_sre.SRE_Pattern object at 0x5577020>**

`field_specifier_NFSC_val` **= ‘(?P<keyword>[a-zA-Z_]\\w*(\\.\\d+)?(\\.[a-zA-Z_]\\w*(\\.\\d+)?)*): (?P<val>[+-]? *(\\**

`field_specifier_NFSC_val_RE` **= <_sre.SRE_Pattern object at 0x5577780>**

`field_specifier_s` **= ‘[a-zA-Z_]\\w*(\\.\\d+)?(\\.[a-zA-Z_]\\w*(\\.\\d+)?)*’**

field_specifier_val = **‘(?P<keyword>[a-zA-Z_]\\w*(\\.\\d+)?(\\.[a-zA-Z_]\\w*(\\.\\d+)?)*): (?P<val>[+-]?(\\.\\d+|\\d+**

identifier = **‘[a-zA-Z_]\\w*’**

key
: Card keyword

keyword_NFSC_val = **“\\’(?P<keyword>[a-zA-Z_]\\w*(\\.\\d+)?(\\.[a-zA-Z_]\\w*(\\.\\d+)?)*): (?P<val>[+-]? *(\\.\\d+|\\d+**

keyword_NFSC_val_comm = **” +\\’(?P<keyword>[a-zA-Z_]\\w*(\\.\\d+)?(\\.[a-zA-Z_]\\w*(\\.\\d+)?)*): (?P<val>[+-]? *(\\**

keyword_NFSC_val_comm_RE = **<_sre.SRE_Pattern object at 0x5579d10>**

keyword_name_RE = **<_sre.SRE_Pattern object at 0x5089af0>**

keyword_val = **“\\’(?P<keyword>[a-zA-Z_]\\w*(\\.\\d+)?(\\.[a-zA-Z_]\\w*(\\.\\d+)?)*): (?P<val>[+-]?(\\.\\d+|\\d+(\\.\\d*)?**

keyword_val_comm = **” +\\’(?P<keyword>[a-zA-Z_]\\w*(\\.\\d+)?(\\.[a-zA-Z_]\\w*(\\.\\d+)?)*): (?P<val>[+-]?(\\.\\d+|\\d+**

keyword_val_comm_RE = **<_sre.SRE_Pattern object at 0x5576a70>**

length = **80**

raw
: Return this card as a normal Card object not parsed as a record-valued keyword card. Note that this returns a copy, so that modifications to it do not update the original record-valued keyword card.

value
: Card value

2.5.3 Free functions

create_card

pyfits.**create_card**(*key=’‘*, *value=’‘*, *comment=’‘*)
: Create a card given the input `key`, `value`, and `comment`. If the input key and value qualify for a `RecordValuedKeywordCard` then that is the object created. Otherwise, a standard `Card` object is created.

  Parameters

  key : str, optional

  The key

  value : str, optional

  The value

  comment : str, optional

  The comment

Returns

card :

Either a `RecordValuedKeywordCard` or a `Card` object.

create_card_from_string

`pyfits.`**`create_card_from_string`**(*input*)

Create a card given the `input` string. If the `input` string can be parsed into a key and value that qualify for a `RecordValuedKeywordCard` then that is the object created. Otherwise, a standard `Card` object is created.

Parameters

input : str

The string representing the card

Returns

card :

either a `RecordValuedKeywordCard` or a `Card` object

upper_key

`pyfits.`**`upper_key`**(*key*)

`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 : int or str

A keyword value that could be an integer, a key, or a `key.field-specifier` value

Returns

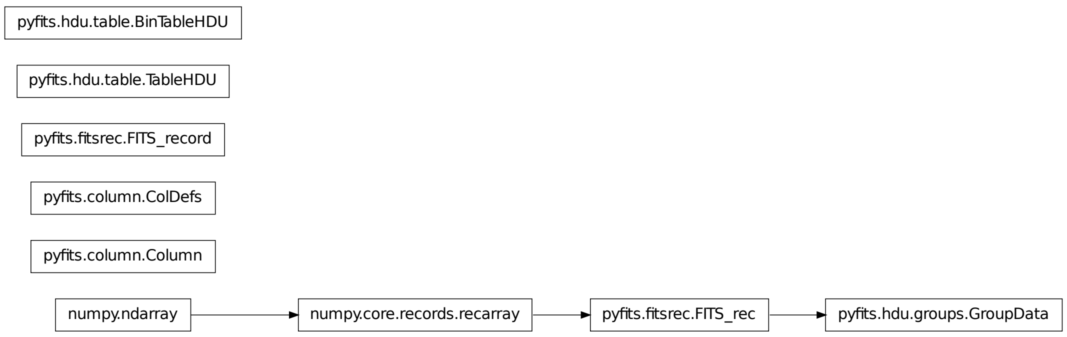
Integer input :

the original integer key

String input :

the converted string

2.6 Tables


pyfits.hdu.table.BinTableHDU
pyfits.hdu.table.TableHDU
pyfits.fitsrec.FITS_record
pyfits.column.ColDefs
pyfits.column.Column
numpy.ndarray
numpy.core.records.recarray
pyfits.fitsrec.FITS_rec
pyfits.hdu.groups.GroupData

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.

`filebytes`()

Calculates and returns the number of bytes that this HDU will write to a file.

Parameters

None :

Returns

Number of bytes :

`fileinfo`()

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

Parameters

None :

Returns

dictionary or None :

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

Dictionary contents:

Key	Value
file	File object associated with the HDU
file-mode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
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.

Parameters

data : str

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] Returns the table's column definitions.

classmethod **`match_header`**(*header*)

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

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

Parameters

fileobj : file object or file-like object

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

checksum : bool

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

ignore_missing_end : bool

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

classmethod **`register_hdu`**(*hducls*)

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

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

TODO: Write about parameters

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

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

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

classmethod `tcreate`(*datafile*, *cdfile=None*, *hfile=None*, *replace=False*)
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 current values in this HDU.

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.

Notes

The primary use for the `tcreate` method is to allow the input of ASCII data that was edited in a standard text editor of the table data and parameters. The `tdump` 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.

tdump(*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 `tdump` method is to allow editing in a standard text editor of the table data and parameters. The `tcreate` method 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.

classmethod `unregister_hdu`(*hducls*)

`update`()

Update header keywords to reflect recent changes of columns.

`update_ext_name`(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)

Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

value to be used for the new extension name

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

`update_ext_version`(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)

Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

value to be used for the new extension version

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

`verify`(*option='warn'*)

Verify all values in the instance.

Parameters

option : str

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

`verify_checksum`(*blocking='standard'*)

Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU CHECKSUM.

blocking: str, optional

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no `CHECKSUM` keyword present

`verify_datasum`(*blocking='standard'*)

Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

blocking: str, optional

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no `DATASUM` keyword present

`writeto`(**args*, ***kwargs*)

Works similarly to the normal writeto(), but prepends a default `PrimaryHDU` are required by extension HDUs (which cannot stand on their own).

`columns`

Works similarly to property(), but computes the value only once.

Adapted from the recipe at http://code.activestate.com/recipes/363602-lazy-property-evaluation

data

Works similarly to property(), but computes the value only once.

Adapted from the recipe at http://code.activestate.com/recipes/363602-lazy-property-evaluation

header

name

tdump_file_format = '\n\n- **datafile: Each line of the data file represents one row of table\n data. The data is outp**

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.

Parameters
None :

Returns
Number of bytes :

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

Parameters
None :

Returns
dictionary or None :

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

Dictionary contents:

Key	Value
file	File object associated with the HDU
file-mode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
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.

Parameters
data : str

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] Returns the table's column definitions.

classmethod `match_header`(*header*)

classmethod `readfrom`(*fileobj*, *checksum=False*, *ignore_missing_end=False*, ***kwargs*)
Read the HDU from a file. Normally an HDU should be opened with `fitsopen()` which reads the entire HDU list in a FITS file. But this method is still provided for symmetry with `writeto()`.

Parameters

fileobj : file object or file-like object

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

checksum : bool

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

ignore_missing_end : bool

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

classmethod `register_hdu`(*hducls*)

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

TODO: Write about parameters

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

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

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

classmethod `unregister_hdu`(*hducls*)

`update`()
Update header keywords to reflect recent changes of columns.

`update_ext_name`(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)
Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

value to be used for the new extension name

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

`update_ext_version`(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)

Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

value to be used for the new extension version

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

`verify`(*option='warn'*)

Verify all values in the instance.

Parameters

option : str

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

verify_checksum(*blocking='standard'*)
Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU CHECKSUM.

blocking: str, optional
"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns
valid : int

- 0 - failure
- 1 - success
- 2 - no `CHECKSUM` keyword present

verify_datasum(*blocking='standard'*)
Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

blocking: str, optional
"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns
valid : int

- 0 - failure
- 1 - success
- 2 - no `DATASUM` keyword present

writeto(**args*, ***kwargs*)
Works similarly to the normal writeto(), but prepends a default `PrimaryHDU` are required by extension HDUs (which cannot stand on their own).

columns
Works similarly to property(), but computes the value only once.

Adapted from the recipe at http://code.activestate.com/recipes/363602-lazy-property-evaluation

data
Works similarly to property(), but computes the value only once.

Adapted from the recipe at http://code.activestate.com/recipes/363602-lazy-property-evaluation

header

name

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. Does not support `theap` yet.

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 commonName list) of a `Column`.

col_name
[str or int] The column name or index to change

attrib
[str] The attribute name

value
[object] The new value for the attribute

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

col_name
[str] The current name of the column

new_name
[str] The new name of the column

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

col_name
[str or int] The column name or index

new_unit
[str] The new unit for the column

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

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

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

Parameters
attrib : str

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

output : file, optional

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

Notes

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

data
What was originally self.columns is now self.data; this provides some backwards compatibility.

2.6.5 `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.

2.6.6 `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`.

x.__init__(...) initializes x; see help(type(x)) for signature

`field`(*key*)

A view of a `Column`'s data as an array.

`columns`

A user-visible accessor for the coldefs. See ticket #44.

2.6.7 `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.

x.__init__(...) initializes x; see help(type(x)) for signature

par (*parname*)
Get the group parameter values.

data

2.6.8 Free 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 ColDefs objects

The data to create a table from.

header : Header instance

Header to be used to populate the non-required keywords.

nrows : int

Number of rows in the new table.

fill : bool

If `True`, will fill all cells with zeros or blanks. If `False`, copy the data from input, undefined cells will still be filled with zeros/blanks.

tbtype : str

Table type to be created (“BinTableHDU” or “TableHDU”).

tdump

pyfits.**tdump** (*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 `tdump` 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.

tcreate

`pyfits.`**`tcreate`**(*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 `tcreate` function is to allow the input of ASCII data that was edited in a standard text editor of the table data and parameters. The tdump 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.

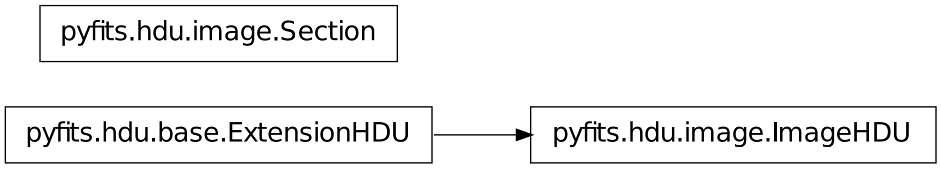
  For column data containing variable length arrays ('P' format), the array data is preceded by the string `'VLA_Length= '` and the integer length of the array for that row, left-justified in a 21-character field, followed by a blank.

  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*)

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.

`add_checksum`(*when=None*, *override_datasum=False*, *blocking='standard'*)

Add the `CHECKSUM` and `DATASUM` cards to this HDU with the values set to the checksum calculated for the HDU and the data respectively. The addition of the `DATASUM` card may be overridden.

Parameters

when : str, optional

comment string for the cards; by default the comments will represent the time when the checksum was calculated

override_datasum : bool, optional

add the `CHECKSUM` card only

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Notes

For testing purposes, first call `add_datasum` with a `when` argument, then call `add_checksum` with a `when` argument and `override_datasum` set to `True`. This will provide consistent comments for both cards and enable the generation of a `CHECKSUM` card with a consistent value.

`add_datasum`(*when=None*, *blocking='standard'*)

Add the `DATASUM` card to this HDU with the value set to the checksum calculated for the data.

Parameters

when : str, optional

Comment string for the card that by default represents the time when the checksum was calculated

blocking: str, optional :

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns

checksum : int

The calculated datasum

Notes

For testing purposes, provide a `when` argument to enable the comment value in the card to remain consistent. This will enable the generation of a `CHECKSUM` card with a consistent value.

`copy`()

Make a copy of the HDU, both header and data are copied.

`filebytes`()

Calculates and returns the number of bytes that this HDU will write to a file.

Parameters

None :

Returns

Number of bytes :

`fileinfo`()

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

Parameters

None :

Returns

dictionary or None :

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

Dictionary contents:

Key	Value
file	File object associated with the HDU
file-mode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
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.

Parameters

data : str

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 **`match_header`**(*header*)

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

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

Parameters

fileobj : file object or file-like object

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

checksum : bool

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

ignore_missing_end : bool

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

classmethod **`register_hdu`**(*hducls*)

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

TODO: Write about parameters

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

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

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

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

Parameters
type : str, optional

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

option : str

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

bscale, bzero : int, optional

User-specified `BSCALE` and `BZERO` values.

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

classmethod **`unregister_hdu`**(*hducls*)

`update_ext_name`(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)
Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters
value : str

value to be used for the new extension name

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

`update_ext_version`(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)

Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

value to be used for the new extension version

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

`update_header`()

Update the header keywords to agree with the data.

`verify`(*option='warn'*)

Verify all values in the instance.

Parameters

option : str

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

`verify_checksum`(*blocking='standard'*)

Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU CHECKSUM.

blocking: str, optional

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure

- 1 - success
- 2 - no `CHECKSUM` keyword present

`verify_datasum`(*blocking='standard'*)
Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

blocking: str, optional
"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns
valid : int

- 0 - failure
- 1 - success
- 2 - no `DATASUM` keyword present

`writeto`(**args*, ***kwargs*)
Works similarly to the normal writeto(), but prepends a default `PrimaryHDU` are required by extension HDUs (which cannot stand on their own).

`ImgCode` = {'uint64': 64, 'uint16': 16, 'int16': 16, 'int64': 64, 'int32': 32, 'float64': -64, 'uint8': 8, 'float32': -32, 'uint32'

`NumCode` = {-64: 'float64', -32: 'float32', 32: 'int32', 8: 'uint8', 64: 'int64', 16: 'int16'}

`data`
Works similarly to property(), but computes the value only once.

Adapted from the recipe at http://code.activestate.com/recipes/363602-lazy-property-evaluation

`header`

`is_image`

`name`

`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`.

`standard_keyword_comments` = {'BITPIX': 'array data type', 'XTENSION': 'Image extension', 'SIMPLE': 'confor

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

data of the image

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 'RICE_1', 'PLIO_1', 'GZIP_1', '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.

filebytes()

Calculates and returns the number of bytes that this HDU will write to a file.

Parameters

None :

Returns

Number of bytes :

fileinfo()

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

Parameters

None :

Returns

dictionary or None :

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

Dictionary contents:

Key	Value
file	File object associated with the HDU
file-mode	Mode in which the file was opened (readonly, copyonwrite, update, append, ostream)
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.

Parameters

data : str

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] Returns the table's column definitions.

classmethod **match_header**(*header*)

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

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

Parameters

fileobj : file object or file-like object

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

checksum : bool

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

ignore_missing_end : bool

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

classmethod `register_hdu`(*hducls*)

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

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

TODO: Write about parameters

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

`run_option`(*option='warn'*, *err_text=''*, *fix_text='Fixed.'*, *fix=None*, *fixable=True*)

Execute the verification with selected option.

`scale`(*type=None*, *option='old'*, *bscale=1*, *bzero=0*)

Scale image data by using `BSCALE` and `BZERO`.

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

Parameters

type : str, optional

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

option : str, optional

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

bscale, bzero : int, optional

user specified `BSCALE` and `BZERO` values.

`size`()

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

classmethod `tcreate`(*datafile*, *cdfile=None*, *hfile=None*, *replace=False*)

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 current values in this HDU.

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.

Notes

The primary use for the `tcreate` method is to allow the input of ASCII data that was edited in a standard text editor of the table data and parameters. The `tdump` 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.

tdump(*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 `tdump` method is to allow editing in a standard text editor of the table data and parameters. The `tcreate` method 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.

classmethod `unregister_hdu`(*hducls*)

`update`()

Update header keywords to reflect recent changes of columns.

updateCompressedData()

Compress the image data so that it may be written to a file.

updateHeader()

Update the table header cards to match the compressed data.

updateHeaderData(*image_header*, *name=None*, *compressionType=None*, *tileSize=None*, *hcompScale=None*, *hcompSmooth=None*, *quantizeLevel=None*)

Update the table header (`_header`) to the compressed image format and to match the input data (if any). Create the image header (`_image_header`) from the input image header (if any) and ensure it matches the input data. Create the initially-empty table data array to hold the compressed data.

This method is mainly called internally, but a user may wish to call this method after assigning new data to the `CompImageHDU` object that is of a different type.

Parameters

image_header : Header instance

header to be associated with the image

name : str, optional

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

compressionType : str, optional

compression algorithm 'RICE_1', 'PLIO_1', 'GZIP_1', 'HCOMPRESS_1'; if this value is `None`, use value already in the header; if no value already in the header, use 'RICE_1'

tileSize : sequence of int, optional

compression tile sizes as a list; if this value is `None`, use value already in the header; if no value already in the header, treat each row of image as a tile

hcompScale : float, optional

HCOMPRESS scale parameter; if this value is `None`, use the value already in the header; if no value already in the header, use 1

hcompSmooth : float, optional

HCOMPRESS smooth parameter; if this value is `None`, use the value already in the header; if no value already in the header, use 0

quantizeLevel : float, optional

floating point quantization level; if this value is `None`, use the value already in the header; if no value already in header, use 16

update_ext_name(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)

Update the extension name associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

value to be used for the new extension name

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

`update_ext_version`(*value*, *comment=None*, *before=None*, *after=None*, *savecomment=False*)
Update the extension version associated with the HDU.

If the keyword already exists in the Header, it's value and/or comment will be updated. If it does not exist, a new card will be created and it will be placed before or after the specified location. If no `before` or `after` is specified, it will be appended at the end.

Parameters

value : str

value to be used for the new extension version

comment : str, optional

to be used for updating, default=None.

before : str or int, optional

name of the keyword, or index of the `Card` before which the new card will be placed in the Header. The argument `before` takes precedence over `after` if both specified.

after : str or int, optional

name of the keyword, or index of the `Card` after which the new card will be placed in the Header.

savecomment : bool, optional

When `True`, preserve the current comment for an existing keyword. The argument `savecomment` takes precedence over `comment` if both specified. If `comment` is not specified then the current comment will automatically be preserved.

`verify`(*option='warn'*)
Verify all values in the instance.

Parameters

option : str

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

`verify_checksum`(*blocking='standard'*)
Verify that the value in the `CHECKSUM` keyword matches the value calculated for the current HDU CHECKSUM.

blocking: str, optional
"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no `CHECKSUM` keyword present

verify_datasum(*blocking='standard'*)

Verify that the value in the `DATASUM` keyword matches the value calculated for the `DATASUM` of the current HDU data.

blocking: str, optional

"standard" or "nonstandard", compute sum 2880 bytes at a time, or not

Returns

valid : int

- 0 - failure
- 1 - success
- 2 - no `DATASUM` keyword present

writeto(**args*, ***kwargs*)

Works similarly to the normal writeto(), but prepends a default `PrimaryHDU` are required by extension HDUs (which cannot stand on their own).

columns

Works similarly to property(), but computes the value only once.

Adapted from the recipe at http://code.activestate.com/recipes/363602-lazy-property-evaluation

compData

Works similarly to property(), but computes the value only once.

Adapted from the recipe at http://code.activestate.com/recipes/363602-lazy-property-evaluation

data

Works similarly to property(), but computes the value only once.

Adapted from the recipe at http://code.activestate.com/recipes/363602-lazy-property-evaluation

header

Works similarly to property(), but computes the value only once.

Adapted from the recipe at http://code.activestate.com/recipes/363602-lazy-property-evaluation

name

shape

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

tdump_file_format = '\n\n- **datafile: Each line of the data file represents one row of table\n data. The data is out**

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 Exceptions and Utility Classes

2.8.1 Exceptions

`VerifyError`

class `pyfits.`**`VerifyError`**

Bases: `exceptions.Exception`

Verify exception class.

x.__init__(...) initializes x; see help(type(x)) for signature

2.8.2 Utility Classes

`Delayed`

class `pyfits.`**`Delayed`**(*hdu=None*, *field=None*)

Bases: `object`

Delayed file-reading data.

`Undefined`

class `pyfits.`**`Undefined`**

Undefined value.

2.9 Verification options

There are 5 options 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.

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. 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.)

3. 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.

4. 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.

5. 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.)

6. 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.

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.

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 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
    ```

20. When releasing a new major or minor version, create a bugfix branch for that version. Starting from the tagged changeset, 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. 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".

22. We also mirror the most recent documentation at pythonhosted.org/pyfits (formerly packages.python.org). The easiest way to do this is to wait until the documentation has been built by Read the Docs (otherwise it is necessary to build the docs yourself) and download it as a zip file. For version 3.2.0 the URL would be:

    https://media.readthedocs.org/htmlzip/pyfits/v3.2.0/pyfits.zip

    (just replace the version part of the URL with the appropriate version).

    Then on the package management page on PyPI (https://pypi.python.org/pypi?%3Aaction=pkg_edit&name=pyfits) locate the documentation upload form and upload the just-downloaded zip file.

23. 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 ... >
    ```