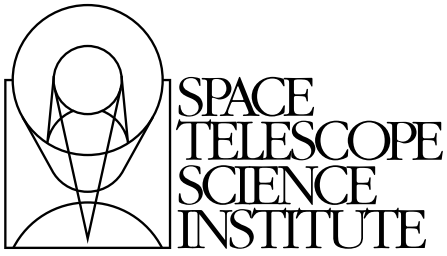
July 2005

PyFITS User's Manual

For PyFITS Version 1.0



Space Telescope Science Institute
3700 San Martin Drive
Baltimore, Maryland 21218
help@stsci.edu

How to Get Started

If you are interested in submitting an HST proposal, then proceed as follows:

- Visit the Cycle 15 Announcement Web page:

`http://www.stsci.edu/hst/proposing`

- Read the Cycle 15 Call for Proposals.
- Read this HST Primer.

Then continue by studying more technical documentation, such as that provided in the Instrument Handbooks, which can be accessed from:

`http://www.stsci.edu/hst/HST_overview/documents`

Where to Get Help

- Visit STScI's Web site at: http://www.stsci.edu
- Contact the STScI Help Desk. Either send e-mail to help@stsci.edu or call 1-800-544-8125; from outside the United States and Canada, call [1] 410-338-1082.

The HST Primer for Cycle 15 was edited by

Diane Karakla, Editor and **Susan Rose**, Technical Editor

based in part on versions from previous cycles, and with text and assistance from many different individuals at STScI.

Send comments or corrections to:
Space Telescope Science Institute
3700 San Martin Drive
Baltimore, Maryland 21218
E-mail:help@stsci.edu

Table of Contents

CHAPTER 1:

Introduction

In this chapter . . .

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 Install PyFITS

PyFITS requires Python version 2.3 or newer. PyFITS also requires the **numarray** module. Information about numarray can be found in:

```
http://www.stsci.edu/resources/software_hardware/numarray
```

To download numarray, 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, readgeis and fitsdiff 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 --local=<install-dir>
setenv PYTHONPATH <install-dir>

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 numarray is not required, but it will help to understand the data structures in PyFITS.

1.2 User Support for PyFITS

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

CHAPTER 2:

A Quick Tutorial

In this chapter . . .

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

```
>>> import pyfits
```

2.1 Read and Update Existing FITS Files

2.1.1 Open 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.

The HDUList has a useful method `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 `close()` method:

```
>>> hdulist.close()
```

The headers will still be accessable after the HDUlist is closed. The data may or may not be accessable depending on whether the data are touched and if they are memory-mapped, see later chapters for detail.

2.1.2 Working with the 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
>>> hdr['targname'] = 'NGC121-a'
>>> hdr[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 `update()` method:

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

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 `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 `keys()` method of the card list:

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

2.1.3 Working with Image Data

If an HDU's data is an image, the `data` attribute of the HDU object will return a numarray object. Refer to the numarray Manual for details of 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.: `hdulist['sci',2]`.

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

```
>>> scidata.shape
(800, 800)
>>> scidata.type()
Float32
```

Since image data is a numarray 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 sub-section of x=11 to 20 (inclusive) and y=31 to 40 (inclusive) is:

```
>>> 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.

Next example of array arithmetics 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. (Note: before Python 2.2.3, the use of "*=" may cause an error, this is fixed in later Python versions.)

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

2.1.4 Working with Table Data

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

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

A numarray 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 its `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 numarray object, we'll have the entire arsenal of numarray tools to use. We can reassign (update) the values:

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

The `info()` method of table data will show the attributes of the record array, many of them may seem esoteric to casual users:

```
>>> tbdata.info()
class: <class 'pyfits.FITS_rec'>
shape: (2,)
strides: (12,)
byteoffset: 0
bytestride: 12
itemsize: 12
aligned: 0
contiguous: 1
buffer: <memory at 0x092dc3d8 with size:0x00000018 held by
object 0x4086eb20 aliasing object 0x00000000>
data pointer: 0x092dc3d8 (DEBUG ONLY)
field names: ['c1', 'c2', 'c3', 'c4']
field formats: ['1Int32', '1a3', '1Float32', '1Int8']
```

2.2 Create New FITS Files

2.2.1 Save Changes

As mentioned earlier, after a user opened a file, made a few changes to either header or data, the user can use the `writeto()` method in `HDUList` 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 `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
```

2.2.2 Create FITS Images from Scratch

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 numarray object for the data part:

```
>>> import numarray
>>> n = numarray.arange(100) # a simple sequence from 0 to 99
```

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:

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

accomplishes the same!

2.2.3 Create FITS Tables from Scratch

To create a table HDU is a little more involved than image HDU, because 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. Say, we have two columns, the first contains strings, and the second contains floating point numbers

```
>>> a1=numarray.strings.array(['NGC1001', 'NGC1002', 'NGC1003'])
>>> a2=numarray.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)
```

Second, 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`. Append it to the `hdulist` we already have:

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

or create a new `HDUList` and go through the same steps as you did for the `ImageHDU`. 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')
```

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.

2.3 Use the 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 specifiy which HDU the user wants to get:

```
>>> 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 arguemnt `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
```

CHAPTER 2:

FITS Headers

In this chapter . . .

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.

2.1 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

numarray. 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.

2.2 The Header Attribute

2.2.1 Value Access and Update

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)
```

2.2.2 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: `add_history()`, `add_comment()`, and `add_blank()`. They are provided because the `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.

2.3 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 in 3.4 below.

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 `.commet` can be changed but not the `.key` attribute.

The `Card()` constructor will check if the arguments given are conforming to the FITS starndard 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 method `fromstring()` can be used.

Cards can be verified by the `verify()` method. The non-standard card c2 in the example below, is flagged by such verification. More about verification in PyFITS will be discussed in Chapter 6.

```
>>> 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.'
```

2.4 Card List

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

The header is basically a list of Cards. This list can be manifested as a `CardList` object in PyFITS. It is accessed via the `ascardlist()` method (or the `.ascard` attribute, for short) of Header. 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
```

2.5 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.

2.6 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

```
>>> 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)
```

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

```
>>> 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
```

keyword name will preserve its cases from its constructor, but when refer to the keyword, it is case insensitive.

CHAPTER 3:

Image Data

In this chapter . . .

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

3.1 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 numarray, 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 numarray data types:.

```
BITPIX      numarray data type

  8         UInt8   (note it is UNsigned integer)
 16         Int16
 32         Int32
-32         Float32
-64         Float64
```

To recap the fact that in numarray 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 numarray 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] = numarray.log(scidata[scidata>0])
```

These examples show the concise nature of numarray operations.

3.2 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.

3.2.1 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 numarray 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 (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.type()
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."
```

3.2.2 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(numarray.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')
```

3.3 Data Section

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()` method can be used to alleviate such memory problem.

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 = numarray.zeros(5000*5000)
>>> for i in range(50):
...     j=i*100
...     k=j+100
...     x1=f[1].section[j:k,:]
...     x2=f[2].section[j:k,:]
...     x3=f[3].section[j:k,:]
...     # use numarray.image's median function
...     output[j:k] = numarray.image.median([x1,x2,x3])
```

Data in each `.section` must be contiguous. Therefore, if `f[1].data` is a 400x400 image, the first part of the following specifications will not work, while the second part will:

```
>>> # These will NOT work, since the data are not contiguous!
>>> f[1].section[:5,:5]
>>> f[1].section[:,:3]
>>> f[1].section[:,2]

>>> # but these will work:
>>> f[1].section[5,:]
>>> f[1].section[5,:10]
>>> f[1].section[6,7]
```

At present, the `section()` method does *not* support scaled data.

CHAPTER 4:

Table Data

In this chapter . . .

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 FITS standard: binary table and ASCII table. Binary table is more economical in storage and faster in data access and manipulation. ASCII table stores 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.

4.1 Table Data as a Record Array

4.1.1 What is Record Array

A record array is an array which contains records (i.e. rows) of heterogeneous data types. Record array is available through the `records` module in the `numarray` library. Here is a simple example of record array:

```
>>> import numarray.records as 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.

4.1.2 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 code 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.

4.1.3 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 Chapter 1:

```
>>> 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
RecArray[
(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.

4.2 Table Operations

4.2.1 Select 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: . [works only on version 0.9.8.4 an later]

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

4.2.2 Merge 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.

4.2.3 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')
```

4.3 Scaled Data in Tables

Table field's data, like an image, can also be scaled. The scaling in 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's 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.

4.4 Create a FITS table

4.4.1 Column Creation

To create a table from scratch, it is necessary to create individual columns first. A `Column` constrctor needs the minimal infomation 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 chapetr. ASCII tables will descussed 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

Note: the current version of PyFITS does not support dim yet.

Here are a few Columns using various combination of these arguments:

```
>>> import numarray as num
>>> from pyfits import Column
>>> counts = num.array([312, 334, 308, 317])
>>> names=num.strings.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) preceeding 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 numarray) 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       '
```

CHAPTER 5:

Verification

In this chapter . . .

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 hold up because of a minor standard violation.

5.1 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 also developed their own sub-standard (dialect?) and some of these become so prevailent and they become de facto standard. One such example is the the long string value and the use of the CONTINUE card (see Chapter 4).

The violation of the standard can happen at different levels of the data structure. PyFITS's verification scheme is developed based on such a hierachiacal levels. Here are the 3 levels of the PyFITS verification levels:

(1) the HDU List.
(2) Each HDU,
(3) Each cardimage in the HDU Header,

At each level, there is a verify() method which can be called at anytime. If the method() is called at the HDL 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 tolerance when reading an FITS file, no verify() is called on input. On output, verify() is called with the most restrictive option as default.

These three levels corresponds 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.

5.2 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 o.utput, the other options listed below can be use

ignore

This option will ignore any FITS standard violation. On output, it will write the HDU List content to the output FITS file, whether or not it is conforming to FITS standard.

The `ignore` option is useful in these situations, for example, (1) An input FITS file with non-standard is read and the user wants to copy or write out after some modification to an output file. The non-standard will be preserved in such output file. (2) A user wants to create a non-standard FITS file on purpose, possibly for testing purpose.

No warning message will be printed out. This is like a silent `warn` (see below) option.

fix

This option wil try to fix any FITS standard violations. It is not always possible to fix such violations. In feneral, 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 priciple 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.

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 lthe same as the `ignore` option but will send warning messages. It will not try to fix any FITS standard violations whether fixable or not.

5.3 Verifications at Different Data Object Levels

We’ll examine what PyFITS’s verification does at the three different elevels:

5.3.1 Verification at HDUList

At the HDU List level, the verification is only for two simple cases:

(1) Verify 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 to the HDU list.

(2) Verify second or later HDU in the HDU list is not a Primary HDU. Violation will not be fixable.

5.3.2 Varification 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 keyword is 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
```

5.3.3 Varification 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 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) unparsable values will be "fixed" as a string.

Here are some examples of fixable card:

```
>>> 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:

```
# create a PrimaryHDU
>>> h=pyfits.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.'
```

CHAPTER 6:

Less Familiar Objects

In this chapter . . .

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,

6.1 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 user interface for ASCII tables and binary tables are basically the same, i.e. the data is in the `.data` attribute and the `field()` method is used to refer to the columns and it returns a numarray or strings

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]
array(
[(10.123000144958496, 37)],
formats=['1a11', '1a5'],
shape=1,
names=['a', 'b'])
>>> hdus[1].data.field('a')
array([  10.12300014,    5.19999981,   15.60999966,    0.        ,
        345.        ], type=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`’).

6.1.1 Create an ASCII Table

To create 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 is more limited than the binary table. It does not allow more than one numerical value in a cell. Also, it only supports a subset of what allowed in the 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 binary table but ’`A3`’ in 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 numarray.strings as chararray
>>> a1 = chararray.array(['abcd','def'])
>>> r1 = numarray.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
array(
[('abc', 11.0, 91),
('def', 12.0, 92),
(' ', 0.0, 93)],
formats=['1a3', '1a14', '1a10'],
shape=3,
names=['abc', 'def', 't1'])
```

6.2 Variable Length Array Tables

FITS standard also supports variable length array tables. The basic idea is that sometimes, it is desirable to have tables whose cells in the same field (column) 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 length 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 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 maimum 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], type=Int16) array([88,  2], type=Int16)
 array([ 1, 88,  3], type=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.

6.2.1 Create Variable Length Array Table

To create 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 which is

included in the numarray module. Here is an example of creating a table with two fields, one is regular and the other variable length array.

```
>>> import numarray.objects as obj

# Define columns
# What's in the parenthesis of the P format is not important, it can be blank
>>> c1 = pyfits.Column(name='var', format='PJ()',\
         array=obj.array([[45., 56], numarray.array([11, 12, 13])]))
>>> c2 = pyfits.Column(name='xyz',format='2I',array=[[11,3],[12,4]])

# the rest is the same as regular table.
# Create the table HDU
>>> tbhdu=pyfits.new_table([c1,c2])
>>> print tbhdu.data
RecArray[
(array([45, 56]), array([11,  3], type=Int16)),
(array([11, 12, 13]), array([12,  4], type=Int16))
]

# 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      '
```

6.3 Random Access Group

Another less familiar data structure supported by FITS standard is the random access group. This convention was established before the binary table extension was introduced. In most cses its use can now be superseded by the binary table. It is mostly used in radio interferometry.

Like Primary HDU, a Random Access Group HDU is always the first HDU of a FITS file. It's 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 NAXIS's) 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)
```

6.3.1 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 `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
```

6.3.2 Data: Group Parameters

The data part of a random access group HDU is, like other HDU's, in the `.data` attribute. It includes both parameter(s) and image array(s).

```
# 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]]]], type=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:

```
# get the parameter names
>>> f[0].data.parnames
['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:

```
# Access group parameter by name or by index
>>> print f[0].data.par(0)[99]
-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 access 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])
```

6.3.3 Data: Image Data

The image array of the data portion is accessable by the `.data` attribute of the data object. A numarray is returned:

```
# image part of the data
>>> 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)
```

6.3.4 Create 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 = numarray.arange(100., 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 = numarray.arange(10)+0.1
>>> pdata2 = 42

# Create the group data object, put parameter names and parameter data
# in lists and 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]
RecArray[
(0.10000000149011612, 42.0, array([[[[ 0.,  1.,  2.,  3.,  4.],
         [ 5.,  6.,  7.,  8.,  9.]]]], type=Float32)),
(1.1000000238418579, 42.0, array([[[[ 10.,  11.,  12.,  13.,  14.],
         [ 15.,  16.,  17.,  18.,  19.]]]], type=Float32))
]
```

CHAPTER 7:

Reference Manual

In this chapter . . .

This chapter lists functions, public classes and their methods, and their arguments in PyFITS.

Index

H

I

M

N

O

P

R

S

T

U

V

W