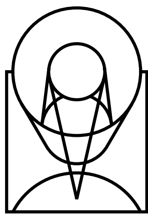
PyFITS Version 0.8
21November 2003

PyFITS User's Manual

Note: PyFITS is still under development. Some of the content in this Manual may change in the future.



SPACE
TELESCOPE
SCIENCE
INSTITUTE

Science Software Group
Engineering and Software Services
3700 San Martin Drive
Baltimore, Maryland 21218

Operated by the Association of Universities for Research in Astronomy, Inc., for the National Aeronautics and Space Administration

Version 1: (14 February 2002)
Version 2: (10 October 2002)
Version 2.1: (21 November 2003)
Written by Perry Greenfield, J.C. Hsu, Warren J. Hack, and Phil Hodge

Copyright © 2003, Association of Universities for Research in Astronomy, Inc. All rights reserved.

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

Table of Contents

CHAPTER 1: Getting Started 1

What is PyFITS? 1

A First Session... 2

Working with Tables 4

Row and Column Selection 5

Help and Feedback 6

User Support 6

Web Page 6

CHAPTER 2: How PyFITS Works 7

Setup PyFITS 7

How to load PyFITS 7

Convention for Usage Examples 7

PyFITS Objects and Methods 8

Header/Data Units 8

Accessing HDUs by Index 8

Accessing HDUs by Name 8

Accessing Multiple HDUs 9

Basic pyfits methods/functions 9

Opening a FITS file 9

FITS file summary method 10

Adding an extension to a FITS file 10

Updating a FITS file 10

Writing out a New FITS file 10

Closing a FITS object 10

FITS objects and I/O 10

Memory Mapping FITS files 11

Creating a new FITS file from scratch 11

FITS file validity 11

Verification checks 12

fix 12

silentfix 12
exception 12
ignore 12
warn 12
Reading non-compliant FITS files 13
Working with Headers 13
Header Objects 13
Getting Detailed Header Information 13
Listing all existing keywords in a header 13
Determining the existence of a particular keyword 14
Accessing a single keyword value 14
Safely returning a keyword value 14
Setting a single keyword value 14
Updating or Adding a Header Keyword 14
Deleting a Header Keyword 15
Images and PyFITS memory usage 15
Memory Conservation with Multiple HDUs 16
Updating an HDU in-place 16
BSCALE and BZERO 16
Working with Binary Tables 17
Interactive Session with Sample Table 17
Record formats 18
Displaying information about a table 18
Get column definitions for a table 18
Get number of rows in a table 19
Access elements of a record (row) 19
Extract column attributes 19
Additional Methods for ColDefs (column definitions) 19
Using String Arrays 19
Creating tables 20
Creating a table from scratch 20
Creating a new table from an existing table 21
How to extend or grow tables 21
Scaled columns 21
Appendix A: Source For Examples 23
Sample Table 23

CHAPTER 1: Getting Started

In This Chapter...

What is PyFITS? / 1
A First Session... / 2
Working with Tables / 4
Help and Feedback / 6

This chapter provides an overview of **PyFITS**'s capabilities and describes some basic usage for the first time users. More details about how PyFITS works will be covered in Chapter 2.

What is PyFITS?

The **PyFITS** module provides a Python tool to allow a user to read, write, and manipulate FITS files. FITS (Flexible Image Transport System) is a portable file standard widely used in the astronomy community to store images and tables. PyFITS provides access to FITS data in any Python environment, including **PyRAF.** PyRAF is a Python interface to and a scripting/programming environment for IRAF tasks.

PyFITS uses the Python paradigm to manage the image/table data and headers. The data from an image or table extension is converted to a **numarray**[1] object, allowing array-based operations on the data. PyFITS is portable to any host architecture and graphics device supported by Python and numarray.

1. numarray is a Python library, a replacement for Numeric with additional capabilities. It provides access to a wider variety of arrays, including numbers, character strings, and records (heterogeneous data types, such as a row in a table).

A First Session...

We'll show a few simple PyFITS examples accessing a multi-extension FITS file. We'll assume the reader has some basic familiarity with Python.

Input Image

For this first session, the sample FITS file `sample.fits` is used. This sample file contains a primary header and 3 extensions: namely, an extension for science data (SCI), an extension for error values (ERR), and an extension for data-quality information (DQ). They are all 1024 by 1024 images of single precision floating-point (SCI and ERR) or short integer (DQ) data type.

Working with Images

1. Start Python in interactive mode and load PyFITS:

```
% python
Python 2.3 (#2, Aug 22 2003, 13:47:10) [C] on sunos5
Type "help", "copyright", "credits" or "license" for
more information.
>>> import pyfits
```

2. The first thing is to open the file with the `open` function:

```
>>> fimg = pyfits.open('sample.fits')
```

The FITS file has now been opened with the default read-only mode. The headers are read in and stored in the Python object `fimg` where `fimg` is an HDUList instance, a Python list-like object with one element for each header/data unit (HDU) in the FITS file. Each element of the list is an object with attributes of header and data, which can be used to access the header keywords and the data. This list is in memory; changes made to `fimg` will not affect the input image unless it was opened in update mode or with memory mapping. Chapter 2 will describe the I/O model for PyFITS and includes a discussion of the memory models for the headers and data.

3. Get a summary of objects in the file:

```
>>> fimg.info()
Filename: sample1.fits
No.    Name         Type        Cards  Dimensions  Format
0     PRIMARY    PrimaryHDU      215  ()           Int16
1     SCI        ImageHDU        215  (1024,1024) Float32
2     ERR        ImageHDU         71  (1024,1024) Float32
3     DQ         ImageHDU         35  (1024,1024) Int16
```

This method reproduces the basic information obtained from the `catfits` task in IRAF. This is just one example of how PyFITS includes functionality usually found in separate IRAF tasks.

4. Examine the primary header. The header attribute is a `Header` instance, another PyFITS object. With its `ascardlist` method, it will list all cards in the header:

```
>>> prihdr = fimg[0].header
>>> prihdr.ascardlist()
```

or

```
>>> fimg[0].header.ascardlist()
```

The second or third command replicates another IRAF task, **imheader**, by printing out the entire header for examination.

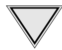
5. Access data in an extension:

```
>>> scidata = fimg[1].data
>>> scidata.shape
(1024, 1024)
```

The object `scidata` points to the data object in the second header-data object in `fimg`, which corresponds to the 'SCI' extension. As a numarray object, it can be used like a Numeric array object in Python.

Alternatively, you can access the extension by its `EXTNAME`:

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



If there are more than one extension with the same `EXTNAME`, `EXTVER` needs to be specified as the second argument, e.g.: `fobj['sci',2]`.

6. Keyword values can be retrieved by using (Python) dictionary notation:

```
>>> exptime = prihdr['exptime']
>>> print exptime
1200.
>>> scihdr = fimg[1].header
>>> photflam = scihdr['photflam']
>>> print photflam
1.3795e-19
```

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.

7. If you know that a keyword is already present in the header, you can update its value using the same notation:

```
>>> prihdr['filename'] = 'sample_flux.fits'
```

8. But if the keyword *might* not be present and you want to add it if it isn't, you can use the `update()` method:

```
>>> prihdr.update('filename', 'sample_flux.fits')
```

9. Operate on the extension's data.

Since image data is a numarray object, we can slice it, view it, and perform mathematical operations on it. Let's convert the image data from counts to flux:

```
>>> scidata *= photflam / exptime
```

This command performs the math on the array in-place, thereby keeping the memory usage to a minimum. (Note: in Python 2.2, the use of "*=" may cause an error, this is fixed in later Python versions.)

10. Write new or modified data and headers to a new FITS file.

```
>>> fimg.writeto('sample_flux.fits')
```

This takes the version of headers and data in memory and writes them to a new file on disk, then closes the file. Further operations could still be performed to the data in memory and written out to yet another different file, all without recopying the original data to (more) memory.

11. Close the input file:

```
>>> fimg.close()
```

That's it! The sample FITS file has been opened, header keywords have been viewed and modified, data from an extension read into memory as a numarray object and manipulated, a new FITS file created, and the original file closed.

Working with Tables

This section describes how to use PyFITS to work with tables. PyFITS uses a model for the table data based on numarray. This model allows PyFITS to work with a table in a couple of usually exclusive ways simultaneously:

- access a table as an array of records (rows in a table)

- access columns (fields) as numarray objects

Row and Column Selection

Data in FITS tables can be read using the same syntax used for image data, but the rows are not stored as simple arrays. Here, we show an example of how PyFITS works with table rows, using the sample table `samp_tab.fits` for this illustration. The sample table contains emission line data, each row containing wavelength, flux, and quality comment. The entire table can be found in the Appendix for reference.

```
Line 1 >>> import pyfits
       >>> intab = pyfits.open('samp_tab.fits')
       >>> tabdat = intab[1].data
       >>> tabcols = intab[1].columns
     5 >>> tabcols.names
       ['wavelength', 'flux', 'quality']
       >>> photcounts = 7.25e+18
       >>> counts = tabdat.field('flux') * photcounts
       >>> print tabdat.field('flux')[:3]
    10 [  2.30739012e-13   3.08164112e-13   4.00046911e-16]
       >>> print tabdat[:2]
       RecArray[
       (1789.832, 2.307390e-13, 'good'),
       (1789.473, 3.081641e-13, 'good')
    15 ]
       >>> x = tabdat.field(0)
```

This example demonstrates several standard operations that would be performed with a FITS table and its data, namely:

- getting the names of the table columns (line 5)
- accessing a column of data as an array (lines 8 and 9)
- display the first two rows (line 11)
- a column (field) can be accessed either by name or by numeric index (line 16)

The data in the table can be accessed either by row or by column. For rows, an object containing all the different types of data from each column is returned. In this example, the wavelength and flux data are floating-point values, but the last column contains strings. A column, however, would be returned by `field()` as a numarray object of one data type.

Help and Feedback

User Support

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

Web Page

The PyFITS web page:

`www.stsci.edu/resources/software_hardware/pyfits`

contains related documents and other resources.

CHAPTER 2:
How PyFITS Works

Setup PyFITS

How to load PyFITS

To use PyFITS, the directory containing PyFITS must be found in your PYTHONPATH. The PyFITS module can be installed in the Python site-packages directory, in which case, Python will look there by default, for the module. Otherwise, the environment variable PYTHONPATH must have the directory explicitly appended.

In a Unix environment, `setenv` can be used to add the PyFITS directory to the PYTHONPATH. If PyFITS was located in `/usr/local/lib`, then the syntax would be:

```
% setenv PYTHONPATH /usr/local/lib/:{$PYTHONPATH}
```

The second instance of PYTHONPATH in this definition insures that any previous settings will be preserved.

Once PyFITS has been added to the Python path, start Python, then import the pyfits module. If you are to work with data, import numarray as well.

```
>>> import pyfits
>>> import numarray
```

Convention for Usage Examples

This section includes examples of the usage of PyFITS methods. These examples will follow what was set up in “A First Session...” on page 2 and “Working with Tables” on page 4 for images and table examples, respectively. This chapter uses the following notation:

```
- fimg = pyfits.open('sample.fits')
- prihdr = fimg[0].header
- scihdr = fimg[1].header
```

- *hdr*: any header object, e.g. *prihdr* or *scihdr*
- *scidata* = fimg[1].data
- *intab* = pyfits.open('samp_tab.fits')
- *tabdat* = intab[1].data
- *tabcols* = intab[1].columns

PyFITS Objects and Methods

Header/Data Units

A FITS file consists of one or more Header/Data Units (HDUs). A simple FITS file consists only of one such Header/Data Unit. Others may have many HDUs. The FITS standard treats the first HDU a bit differently than the rest and calls it the Primary HDU. Any subsequent HDU is called an Extension HDU. When a FITS file is opened in PyFITS, it returns an HDUList object which has list-like properties, but can only hold HDU objects.

Accessing HDUs by Index

- ***Syntax***: fimg[*index*]
- ***Example***: fimg[0]

Like Python lists, HDUList is zero-indexed. If `fimg` is a FITS object, then `fimg[0]` is the primary HDU, `fimg[1]` is the first extension HDU, and `fimg[-1]` is the last HDU.

Accessing HDUs by Name

- ***Syntax***: fimg[*extname*,*extver*]
- ***Example***: fimg['PRIMARY']

The above example shows how to access the primary HDU by name. It is equivalent to `fimg[0]`.

In addition, we can access an *extension* HDU by the value of the keyword EXTNAME.

- ***Example***: fimg['SCI']

This usage will find the extension HDU with `EXTNAME='SCI'`. The second argument `extver` is optional if each EXTNAME has unique value, but is required if there are more than one extension HDU's with the same EXTNAME.

- ***Example***: fimg['sci',2]

This will return the first Extension HDU with `EXTNAME='SCI'` and `EXTVER=2`. Also note that the EXTNAME specification is case-insensitive.

Accessing Multiple HDUs

- ***Syntax***: `fimg[`*range*`]`
- ***Example***: `fimg[`*1:6*`]`

Any `HDULists` object behaves like a Python list, so a user can slice, replace, insert, or append HDU objects like Python lists.

Only HDU objects (or objects that subclass the HDU class) may be placed into HDULists. Attempts to insert, replace or append other kinds of objects will raise an exception.

Basic pyfits methods/functions

This section describes the methods/functions used for working on FITS files as a single object. They include opening, writing, and closing a FITS file, getting information on the contents of a FITS file, and adding an HDU to the file. Table 2.1 lists basic methods used to work with FITS files.

Table 2.1: Basic pyfits methods/functions

Method name	Action
	PyFITS function and class(es)
`open(`*filename*`)`	open a FITS file *filename,* and return a HDUList object
`HDUList()`	instantiate (start/create) an HDUList
	HDUList Methods
`info()`	Print out a summary of contents for a FITS file
`append(`*hdu*`)`	add a header data unit *hdu* as an element to the HDUList
`flush()`	update the associated FITS file on disk with the current version of HDUList in memory
`writeto(`*new_file*`)`	write the HDUList object in memory to a file named *new_file*
`close()`	close the file associated with the HDUList object

Opening a FITS file

- ***Syntax***: `fimg = pyfits.open(`*filename, mode="copyonwrite", memmap=0*`)`

An HDUList is instantiated (or created) from an existing FITS file by using this PyFITS library function. There are 4 different file I/O modes.

- *readonly and copyonwrite*`(default) - do not allow the file on disk to be modified. If memory mapping is not`

`used (memmap=0), these two modes are equivalent. If memory mapping is used, readonly will not allow the data values being changed; copyonwrite will, but these changes cannot be written back to the original disk file.`

- *update* - allow the file on disk to be modified
- *append* - open the file on disk to allow new HDUs to be appended or create a new FITS file from scratch. It, however, will not allow modifications to HDUs that are already present in the file when it is opened.

Also see “FITS objects and I/O” on page 10 for related issues.

FITS file summary method

- ***Syntax***: `fimg.info()`

This method prints information about the HDUs contained in the HDUList.

Adding an extension to a FITS file

- ***Syntax***: `fimg.append(`*hdu*`)`

This method appends an HDU to the existing HDUList in memory. This change will only be written to disk when this HDUList object is closed with the `close` method or updated with the `flush` method.

Updating a FITS file

- ***Syntax***: `fimg.flush()`

This method updates the file on disk with the current version of the HDUList object `fimg` in memory.

Writing out a New FITS file

- ***Syntax***: `fimg.writeto(`*new_file*`)`

Write an existing HDUList in memory to a new file on disk.

Closing a FITS object

- ***Syntax***:`fimg.close()`

This method will close the file on disk. If the file was opened in `update` or `append` mode, it will first apply any changes of the HDUList to the file on disk by calling the `flush` method.

FITS objects and I/O

Except in the case of memory mapping, FITS objects are treated as a memory entity. Regardless of how they were created, once they exist, they maybe be modified in memory (even if the file was opened read-only).

No `write` mode in `open()`! Observant readers may have noticed that there is no ‘write’ mode. This is because any HDUList (including those not associated with files) may be written to a new file by using the `writeto`() method. This includes HDULists opened in any mode.

Memory Mapping FITS files

- ***Syntax***: `fimg = pyfits.open(`*filename, mode, memmap=1*`)`

One may memory map FITS files by setting the *memmap* argument in the open function to a nonzero value. Memory mapped files may only be opened in `readonly`, `copyonwrite`, or `update` modes. Data opened in `copyonwrite` mode may be modified but the modifications will not appear in the original opened disk file. Changes to memory-mapped data in `update` mode are not guaranteed to appear in the file until a flush is done or the HDUList is closed. Changes to headers are not guaranteed to appear in the file until flushed or closed.

Creating a new FITS file from scratch

- ***Syntax***: `hdulist = pyfits.HDUList()`

Usually, an existing FITS file will serve as the initial copy of the headers and data for the HDUList. One way to create a FITS file from scratch is to simply create an empty HDUList. It can then be populated, either by appending HDUs copied from existing FITS files or by creating HDUs from scratch. The first HDU added must be a PrimaryHDU object in order to generate a valid FITS file.

Example 2.1: Example

```
fitsobj = pyfits.HDUList()
# create Primary HDU with minimal header keywords
hdu = pyfits.PrimaryHDU()
# add a 10x5 array of zeros
hdu.data = numarray.zeros((10,5), type= numarray.Float32)
fitsobj.append(hdu)
# save to a file, the writeto method will make sure the required
# keywords are conforming to the data
fitsobj.writeto('myzeros.fits')
```

FITS file validity

- ***Syntax***: `hdulist.verify()`

HDULists objects do not check for self consistency, including headers and data, at the time changes are made to its content. It is possible to construct FITS objects that would not be legal FITS as they exist in memory. For example, one may create an HDUList with only a table HDU. That would be illegal since a table HDU may never be the first HDU. Or

one may change the data array associated with an HDU so that its size, shape, or type is inconsistent with the information in its header (or, less likely, change the header information to be inconsistent with the data).

Verification checks

Consistency checks are performed when the data are written to files or when the `verify()` method is called for HDUs. So, calls to `verify()`, `flush()`, `writeto()`, or `close()` will result in the HDUs being examined for consistency with the FITS standard.

First of all, on output, an HDU's data attributes are checked against its header. The data will take precedence, e.g. if the data dimension is different from what is indicated by the `NAXIS` keywords, the keywords will be modified to agree with the data. This checking includes `NAXIS`, `NAXISi`, and `BITPIX`. Similarly, on output, the keyword `EXTEND`, in the primary HDU is always fixed, regardless of mode.

The verification modes are:

fix

Sensible changes are made to the HDUList to force it to be consistent. Changes will include:

- If a TableHDU or an ImageHDU is the first element in an HDUList, it will cause a basic Primary HDU to be inserted (with no data)
- If the required keywords are missing or out of order, it will fix them.

All changes are reported as messages as they are being performed during the verification process.

silentfix

Same as Fix, except that no messages are printed.

exception

Any inconsistency will cause an exception.

ignore

Inconsistencies are not checked. ONLY USE THIS MODE IF YOUR GOAL IS TO WRITE ILLEGAL FITS FILES FOR TESTING PURPOSES!

warn

Print out what is inconsistent. This is only useful when the `verify` method is called (since that method does not attempt to make any changes). In attempts to write, this mode is identical to `exception`.

Only headers will be 'fixed' when inconsistencies are detected (and the mode is either `fix` or `silentfix`). Data are never changed during verification.

Reading non-compliant FITS files

There are many FITS files that are not strictly compliant with the FITS standard. It is our desire to be able to read all reasonable data even if it does not strictly comply with the standard. Please inform us of examples where PyFITS has difficulty handling and we will address such cases.

Working with Headers

Each FITS file contains one PRIMARY header, optionally with an associated image array, and may contain several other headers (one in each extension HDU) as well. The methods described in this section provide the means for working with the header keywords

Header Objects

FITS headers consist of a set of 'card images.' Each card contains 80 ASCII characters. Most of these cards consist of keyword/value pairs, with an optional associated comment. There are also COMMENT and HISTORY cards. PyFITS header objects present two means to access its content. One is convenient for most usages, and the other gives full control over all aspects of the contents of a header card. Header objects present a dictionary-like (as well as list-like) interface that allows easy access to keyword values.

Getting Detailed Header Information

- ***Syntax***: *cardlist* = hdr.ascardlist()

To obtain detailed information from a header (e.g., the location of a keyword or its comments) or to control the contents of a header (e.g., where new keywords go or to set comments), use the cardlist interface.

The *cardlist* object gives full control over the header, but is less convenient to use, particularly interactively. This object contains the entire header as formatted in the FITS file itself and therefore can be used to manually inspect or search the header for information.

Listing all existing keywords in a header

- ***Syntax***: hdr.items()

This header method will return a list of tuples containing the keyword and value from every card in the header.

Determining the existence of a particular keyword

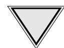
- ***Syntax***: `hdr.has_key(`*keyword*`)`
- ***Example***: `scihdr.has_key('naxis1')`

It will return 1 if the keyword exists in the header, or 0 otherwise.

Accessing a single keyword value

- ***Syntax***: `hdr[`*keyword*`]`
- ***Example***: `prihdr['naxis']`

This syntax allows the user to print out the value of a single keyword. If the keyword does not exist, a KeyError exception will be raised.



PyFITS is case-insensitive towards keyword names.

Safely returning a keyword value

- ***Syntax***: `val = hdr.get(`*keyword*`, `*default_value*`)`

One may use the `get` method as an alternative to simple indexing. It has the advantage of returning the specified default value if the keyword is not present, instead of raising an exception.

- ***Example***: `val = prihdr.get('texptime',600.)`

Will set `val` to the value of TEXPTIME, or to the default value of 600. if TEXPTIME is not present.

Setting a single keyword value

- ***Syntax***: `prihdr[`*keyword*`] = `*value*
- ***Example***: `prihdr['exptime'] = 1200.`

One cannot add new keywords (but see below) this way. This syntax is for existing keywords only.

Updating or Adding a Header Keyword

- ***Syntax***: `prihdr.update(`*keyword*, *new_value*, *comment*`=new_comment,` *after*`=keyword,` *before*`=keyword)`

This method allows the user to update the value of an existing keyword or add a card with a new keyword to the header. The `before` and `after` optional parameters control the placement of the new keyword in the header, and will be ignored for an existing keyword.

- ***Example***: `prihdr.update('TEMP2', 42, comment="CCD2 temperature")`

The example shows how to add the TEMP2 keyword with value of 42 to the primary header. If before and after are not specified, the new card will be added at the bottom of the header.

- ***Example***: prihdr.update('TEMP2', 42, comment="CCD2 temperature", after="TARGET")

This will add the TEMP2 keyword after the TARGET keyword in the primary header.

Deleting a Header Keyword

- ***Syntax***: del hdr[*keyword*]
- ***Example***: del scihdr['texptime']

Delete the TEXPTIME keyword from the science extension header.

Note that all header cards can be accessed by numeric indexing and most of them by keyword name. So prihdr[0] works as well as prihdr['simple']. If the header has two or more keywords with the same name (e.g. HISTORY), referencing by name will only access the first one.

Images and PyFITS memory usage

Astronomical FITS files can be quite large. Some will not fit into the memory. PyFITS is developed to provide a means of minimizing memory usage. The most important is allowing memory-mapped data. If the memmap option is not selected, it is necessary to read the entire data of a *given* HDU into memory to access it. But there are means to avoid reading in all extensions simultaneously.

When a FITS file is opened in any mode, only the headers are read into memory. Data of a given HDU are read into memory only if there is an attempt to access it. For memory-mapped dada, memory space is needed only when the data are *scaled*, i. e. BSCALE != 1 or BZERO != 0

Example 2.2: Reading data into memory.

```
>>> fitsobj = pyfits.open("mydata.fits", "readonly")
# After the open, only headers, but not data, are read into
memory
>>> n1 = fitsobj[1].header['NAXIS1']
>>> d = fitsobj[1].data # data of 1st extension is read into
memory
```

HDULists also has a readall() method which forces all the data to be read into memory. But if one opens a file and only accesses the data of one extension, only data for that extension are read into memory.

Memory Conservation with Multiple HDUs

If one must sequentially process a large multi-extension file and produce a new FITS file with processed images, the technique utilized in Example 2.3 is recommended to conserve memory usage.

Example 2.3: Memory conservation with multiple HDUs.

```
>>> infits = pyfits.open('input.fits')
>>> outfits = pyfits.open('output.fits'. "append")
>>> for hdu in infits:
...    # simple copying of input hdu to output
...    outfits.append(hdu)
...    outfits[-1].data = 2*hdu.data # double image values
...    outfits.flush()               # write appended HDU
...    hdu.data = None         # free memory in input HDU
...    # free memory in output fits object
...    outfits[-1].data = None
>>> outfits.close()
```

In this manner, memory is only being used for one extension at a time. With memory mapping, these sorts of manipulations will not be necessary.

Updating an HDU in-place

If one wants to update an HDU in place, particularly if the header or data will not change in size, Example 2.4 illustrates how that can be done with PyFITS.

Example 2.4: Working with an HDU in-place.

```
>>> fitsobj = pyfits.open('mydata.fits','update')
>>> hdu = fitsobj['SCI', 2]
>>> hdu.data = 2*hdu.data
>>> fitsobj.close()
```

BSCALE and BZERO

Image HDUs that have BSCALE with a value other than 1 and a BZERO with a non-zero value will result in the scaling of the data when the data are read from the file. If the file is memory mapped this means the data will be copied into memory (and thus the benefits of memory mapping do not apply to such HDUs). After BSCALE and BZERO are applied to data when they are read into memory, these two keywords are removed from the header. If a user wishes to have BSCALE and/or BZERO in the final output FITS file, the method `scale()` can be used for any image HDU object.

Working with Binary Tables

The data in binary tables (with fixed record sizes) appear as a special kind of array called record array. Record arrays provide two different ways of accessing table data: by rows or columns. The advantage of record arrays is that these two different ways of accessing the data can be used together without requiring extra copying of data. The data remains (with a few exceptions to be mentioned later) in the same way it does in a FITS file, that is, as a repeated series of rows.

Record arrays are inherited from the same base class, `NDArray`, as `numarrays`, and thus can be accessed with all mechanisms that `NDArray` may have. In particular, they may be indexed and sliced in exactly the same way (including use of index arrays). But record arrays also provide methods that yield numarrays for columns of a table. These numarrays are not copies of the data in the table, but rather a numarray view of exactly the same data in the table. On the other hand, this means the record array's data structure is fixed. In order to modify it, such as deleting a row or a column, a new table has to be made

Interactive Session with Sample Table

This session shows how to use PyFITS on a binary table, including:

- opening the table
- accessing the binary table data
- viewing the data in a record (row)
- extracting or viewing the data of a column

This example is based on the table whose columns and data are given in Table A.1 on page 23. More methods for working with tables are described later in this chapter.

```
>>> import pyfits
>>> tab = pyfits.open('samp_tab.fits')
>>> tabhdu = tab[1]
>>> tabdat = tabhdu.data
>>> # print first record
>>> print tabdat[0]
(1789.83203125, 2.3073901188157275e-13, 'good')
>>> # print every other record starting at 3rd
>>> print tabdat[2::2]
RecArray[
(1789.114013671875, 4.0004691119440762e-16, 'dead'),
(1788.39599609375, 4.8213738773963954e-13, 'good'),
(1787.678955078125, 3.9276061244465643e-13, 'good'),
(1786.9610595703125, 8.2633308832666397e-13, 'good'),
(1786.2430419921875, 2.2820289494879242e-12, 'good'),
```

```
(1785.5250244140625, 2.2860970928795243e-12, 'good'),
(1784.8079833984375, 8.6380837378829955e-13, 'good'),
(1784.0899658203125, 3.3133559914841348e-13, 'good')
]
>>> # obtain a numarray view of the wavelength column
>>> wavelength = tabdat.field('wavelength')
>>> print wavelength
[ 1789.83203125  1789.47302246  1789.11401367  1788.75500488
  1788.39599609  1788.0369873   1787.67895508  1787.31994629
  1786.96105957  1786.60205078  1786.24304199  1785.8840332
  1785.52502441  1785.16699219  1784.8079834   1784.44897461
  1784.08996582  1783.73095703]
>>> # change the first wavelength
>>> wavelength[0] = 1216.
>>> # note change in record view!
>>> print tabdat[0]
(1216.0, 2.3073901188157275e-13, 'good')
```

Record formats

What characters record arrays use to describe records formats is different than that used by FITS (they are setup to handle any numarray type as well as chararrays). The mapping between the two is:

FITS	record array
L	b1
B	u1
I	i2
E	f4
D	f8
J	i4
C	c8
M	c16
A	a

Displaying information about a table

Get column definitions for a table

- ***Syntax***: `tabcols = tabhdu.get_coldefs()` or
- `tabcols = tabhdu.columns`

The `ColDefs` object can be instantiated using one of these syntax. This object contains information about the columns as derived from the header,

including: names, formats, units, etc.. These attributes contain the values for all the columns as a list for each attribute.

Get number of rows in a table

- ***Syntax***: `rows = len(tabdat)`

Access elements of a record (row)

- ***Syntax***: `tabdat[`*row*`].field(`*colname*`)` or `tabdat[`*row*`].field(`*index*`)`

A single element of a record array is a `Record` object. Fields of a `Record` object may be accessed by position or by name.

Extract column attributes

- ***Syntax***: `tabcols.info(`*attribute_name*`)`

Column names are in an attribute of the `ColDefs` object and a list of them can be printed using this method. For example, the sample table has columns named `WAVELENGTH`, `FLUX`, and `QUALITY`. The characteristics for those columns can be accessed:

Example 2.5: Accessing column attributes

```
>>> tab = pyfits.open('samp_dat.fits')
>>> cols = tab[1].columns # same as tab[1].get_coldefs()
>>> cols.info('names')
name:
    ['wavelength', 'flux', 'quality']
>>> cols.info('formats')
format:
    ['f4', 'f4', 'a8']
>>> # Or use cols.names or cols.formats to get the actual
list.
```

Additional Methods for ColDefs (column definitions)

`ColDef` objects also have the following methods: `del_col()`, `change_name()`, `change_unit()`, and `add_col()`.

Using String Arrays

- ***Syntax***: `carr = tabdat.field(`*char_col_name*`)`

Since tables may contain character columns, the string array in numarray is used to handle such columns. Unlike Python strings, string arrays contain items of fixed size. Like NDArrays, they may be indexed and sliced the same way. The sample FITS table has a character column named 'QUALITY' with a width of 8 characters. The character array can

be accessed with the syntax listed above, while Example 2.6 demonstrates how to print individual items from the string array.

Example 2.6: Printing string array members

```
>>> carray = tab.field('quality')
>>> print carray
['good', 'good', 'dead', 'good', 'good', 'good', 'good',
     'good', 'good', 'good', 'good', 'saturate', 'good', 'good',
     'good', 'good', 'good', 'good']
>>> print carray[1:3]
['good', 'dead']
>>> carray[0] = "warm"
>>> print carray[0]
warm
```

Creating tables

– ***Syntax***: `newtab = pyfits.new_table(`*`input, header`*`=None,` *`nrows`*`=0,` *`fill`*`=0,` *`tbtype`*`="BinTableHDU")`

Binary table HDUs may be created using this method. Specific attributes of the new table can be defined upon creation through the use of the parameters:

- **input**
  a list of Column objects or a ColDefs object
- **header**
  header to be used to populate the non-required keywords
- **nrows**
  number of rows for table (default is to take the largest array found in input)
- **fill**
  if fill=0 (default), copy the data from input, otherwise fill all cells with zero or blank
- **tbtype**
  type of table to be created, only BinTableHDU (default) is supported now

Note that the arrays used to construct the new table are referred to by a Column list or `ColDefs` object and it is in constructing either one of these that the data are provided for the new table. The arrays so provided are copied to a single data buffer allocated for the new table. Tables, once created, like other NDArrays, are not resizable. Expanding a table requires creating a new table.

Creating a table from scratch

One may obtain column definitions from an existing table, or use the `Column` object to create a table from scratch, as shown in Example 2.7.

Example 2.7: Creating a table from scratch

```
>>> targets = chararray.array(['M51','NGC4151','Crab
Nebula','Beta Pictoris'])
>>> col1 = pyfits.Column(name='targname',format='13A',
array=targets)
>>> col2 = pyfits.Column('RA',format='E', unit='hours',
array=numarray.array([1.,2.,3.,4.]))
>>> col3 = pyfits.Column('DEC','E','degrees',array=
numarray.arange(4))
>>> tabhdu = pyfits.new_table([col1, col2, col3], nrows=100)
# write to a FITS file, remember that table cannot be at
# the primary HDU, so it is necessary to create a minimal
# primary HDU
>>> hdulist = pyfits.HDUList([pyfits.PrimaryHDU(), tabhdu])
>>> hdulist.writeto('newtable.fits')
```

Creating a new table from an existing table

One may wish to extend a table definition in the creation of a new table, for example by adding a new column as shown in Example 2.8.

Example 2.8: Creating a new table by adding a column to an existing one

```
>>> cdefs = tabhdu.get_coldefs()
>>> cdefs.add_col(col1)
>>> outtabhdu = pyfits.new_table(cdefs)
```

How to extend or grow tables

The user can not only access information about a table, but also change the composition of the table using fairly simple syntax. Example 2.9 shows how a new table can be created from an old one with additional rows.

Example 2.9: Adding rows to a table.

```
>>> rows = len(tabdat)
>>> cdefs = tabhdu.get_coldefs()
>>> # copy data to larger table
>>> outtabhdu = pyfits.new_table(cdefs, nrows=rows+100)
>>> outtabhdu.data.field(0)[rows:] = 3 # add data for each
column
```

Scaled columns

In most cases, record array data are taken directly from the table data buffer. But Boolean columns, columns in an FITS ASCII table, and columns with TSCAL != 1 or TZERO != 0, are scaled and the scaled numarrays need extra separate memory space. Users only see and interact with these scaled arrays. If any of these scaled arrays are modified, the original (unscaled) data are not updated until being written to disk file.

Appendix A:
Source For Examples

In The Appendix...

Sample Table / 23

Sample Table

Table A.1: Sample Table for Examples

WAVELENGTH	FLUX	QUALITY
1789.832	2.307390E-13	good
1789.473	3.081641E-13	good
1789.114	4.000469E-16	dead
1788.755	4.908227E-13	good
1788.396	4.821374E-13	good
1788.037	5.600838E-13	good
1787.679	3.927606E-13	good
1787.32	5.556854E-13	good
1786.961	8.263331E-13	good
1786.602	1.577337E-12	good
1786.243	2.282029E-12	good
1785.884	2.358329E-12	saturate
1785.525	2.286097E-12	good
1785.167	1.517201E-12	good
1784.808	8.638084E-13	good
1784.449	5.023870E-13	good
1784.09	3.313356E-13	good
1783.731	2.677191E-13	good