PyFITS Version 0.7.5
Second Edition
10 October 2002

PyFITS User’s Manual

This Preview Version contains preliminary examples and information regarding a Python module which is still under development. The layout and content of the final User’s Manual may change significantly from what is indicated in this version.
Table of Contents

CHAPTER 1: Getting Started
What is PyFITS? ................................................................. 1
A First Session ................................................................. 2
Working with Tables .......................................................... 5
Row and Column Selection ................................................. 5
Help and Feedback ............................................................ 6
User Support ...................................................................... 7
Web Page ............................................................................ 7

CHAPTER 2: How PyFITS Works
PyFITS Components and Terminology ......................... 9
Quick Introduction to Using Objects ............................... 9
Setup PyFITS ..................................................................... 10
How to load PyFITS .......................................................... 10
Convention for Usage Examples ....................................... 10
PyFITS Objects and Methods .......................................... 11
Header/Data Units ............................................................. 11
Accessing HDUs by Index 11
Accessing HDUs by Name 11
Accessing Multiple HDUs 12
Basic pyfits methods/functions ...................................... 12
Opening a FITS file 12
FITS file summary method 13
Adding an extension to a FITS file 13
Updating a FITS file 13
Writing out a New FITS file 13
Closing a FITS object 13

FITS objects and I/O .......................................................... 14
No write mode 14
Memory Mapping FITS files .......................................................... 14
Creating a new FITS file from scratch ........................................ 14
FITS file validity ....................................................................... 15
   Verification checks 15
      fix (default) 15
      silentfix 16
      exception 16
      ignore 16
      warn 16
   Method of Correction 16
Reading non-compliant FITS files ............................................ 16

Working with Headers .................................................................. 16
   Header Objects ...................................................................... 17
      Getting Detailed Header Information 17
      Listing all existing keywords in a header 17
      Determining the existence of a particular keyword 17
      Accessing a single keyword value 17
      Setting a single keyword value 18
      Safely returning a keyword value 18
      Updating or Adding a Header Keyword 18
      Deleting a Header Keyword 18

Images and PyFITS memory usage ............................................ 19
   Memory Conservation with Multiple HDUs ............................. 19
   Updating an HDU in-place .................................................... 20
   BSCALE and BZERO ............................................................ 20

Working with Binary Tables ..................................................... 20
   Interactive Session with Sample Table .................................... 21
   Record formats .................................................................... 22
   Displaying information about a table ..................................... 22
      Getting column definitions for a table 22
      Getting number of rows in a table 22
      Accessing elements of a record (row) 22
      Extracting column attributes 22
   Additional Column Methods ............................................... 23
   Using Chararrays ................................................................. 23
   Creating tables .................................................................... 24
      Creating a table from scratch 24
      Creating a new table from an existing table 25
   How to extend or grow tables ............................................... 25
   Scaled columns .................................................................... 25
Appendix A: Source For Examples ................................................. 27

Sample Table ................................................................................. 27
4 Contents
This chapter provides an overview of PyFITS’s capabilities and describes some basic techniques for using the interface for the first time users. More details about how PyFITS works and how the software can be used to work with various types of FITS files will be covered in Chapters 2.

What is PyFITS?

The PyFITS module provides a Python user to read, write, and manipulate FITS files, whether they are tables, images, or a combination of both. It 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 does not require PyRAF though.

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 object, allowing array-based operations on the data. The design of PyFITS follows object-oriented methodology and provides an object-oriented view of the data itself. As a Python application, PyFITS is portable to any host architecture and graphics device supported by Python and numarray, and allows the headers and data to be manipulated entirely within Python.

1. numarray is a replacement for Numeric which provides access to a wider variety of data types and representations.
A First Session...

We’ll show examples of PyFITS to access a multi-extension FITS file. Each step for this first session will be shown as it would be done within Python/PyRAF, along with an explanation of its purpose and result. Subsequent examples will simply list and explain PyFITS commands.

Input Image

For this first session, the sample FITS file `sample1.fits` is used. This sample file is a multi-extension image with the following extensions (as given by the IRAF task `catfits`):

<table>
<thead>
<tr>
<th>EXT#</th>
<th>FITSNAME</th>
<th>FILENAME</th>
<th>EXTVE DIMENS</th>
<th>BITPI OBJECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hr_shd.fits</td>
<td>sample.fits</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>IMAGE</td>
<td>sci</td>
<td>1024x1024</td>
<td>-32</td>
</tr>
<tr>
<td>2</td>
<td>IMAGE</td>
<td>err</td>
<td>1024x1024</td>
<td>-32</td>
</tr>
<tr>
<td>3</td>
<td>IMAGE</td>
<td>dq</td>
<td>1024x1024</td>
<td>16</td>
</tr>
</tbody>
</table>

This listing demonstrates the structure of a typical multi-extension FITS file. It contains a PRIMARY header and 3 extensions related to the actual image: namely, an extension for science data (SCI), an extension for data-quality information (DQ), and an extension for error values (ERR).

Working with an Image

1. Start Python in interactive mode and load PyFITS:

   ```
   % python
   Python 2.0 (#1, Oct 27 2000, 09:38:44) [C] on sunos5
   >>> import pyfits
   ```

2. Tell PyFITS what FITS file will be read and the name of the HDUList object generated by the 'open' function:

   ```
   >>> fimg = pyfits.open('sample1.fits')
   ```

   The FITS file has now been opened with the default read-only mode. The headers have been read in and stored in the Python object `fimg`. 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
   ```
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.

All the basic methods for working with Python lists are available with *fimg*, as the FITS object builds on the (Python) list behavior to provide an intuitive way to work with the components of the FITS files. This allows the user to treat each extension in a FITS file as a separate object in a Python list, including adding, sorting and deleting extensions as necessary.

4. Examine the primary header:

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

or

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

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

---

*prihdr* serves strictly as a shorthand, simply making it easier to reference the primary extension’s header later. It refers to the same header and is NOT a copy.

5. Access data in an extension:

```python
>>> 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 an numarray object, it can be used like a Numeric array object in Python.

Alternatively, you can access the extension by its EXTNAME:

```python
>>> 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. The value for any keyword 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

   Keyword access with PyFITS is case-insensitive, allowing the user to ignore case when specifying the keyword.

7. The value for any keyword can also be set by using the same notation:
   >>> prihdr[‘TEXPTIME’] = 1200.

8. A keyword can be added to a header using the update() method:
   >>> prihdr.update(‘PHOTFLAM’, 1.3795E-19)

In this case, we simply added the value of PHOTFLAM to the primary header.

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 will be fixed in future Python versions.)

10. Write new data and updated 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 the 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:

```python
>>> 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, a replacement for Numeric with additional capabilities. 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 (a "row" in a table)
- access columns 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. Figure 1.1 shows 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.
Figure 1.1: Reading a FITS Table

This example demonstrates many of the standard operations that would be performed with a FITS table and it’s 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 selected either by row or by column simply by specifying the slice of the entire table that you want. In the case of rows, an object containing all the different types of data from each column is returned. In this example, the wavelength and flux data were real values, but the last column contains strings. A column, however, would be returned as a numarray object.

Help and Feedback

With PyFITS, we try to provide a simple and powerful interface to FITS files in the Python environment. Feedbacks are welcome on how PyFITS and it’s documentation could be improved. Suggestions on methods which users find necessary for their work would allow us to determine the minimum set of features in PyFITS without cluttering the interface.

We urge users to use the following contact points to send in reports on limitations, bugs, and revelations of how well PyFITS performs. All comments will be critically evaluated and applied as appropriate.
User Support

If you have a question about, comment on, or problem using 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 (http://stsdas.stsci.edu/pyfits) contains related documents and other resources.
CHAPTER 2: How PyFITS Works

This chapter explains how to use PyFITS in greater details.

PyFITS Components and Terminology

PyFITS has been written as an object-oriented module but requiring minimal knowledge of object oriented programming to use it. However, it does help to be familiar with some object-oriented terminology.

Using the module is quite straightforward if one learns a few basic terms and means of using objects.

Quick Introduction to Using Objects

A class defines a type of object. Objects are specific instances of such classes (just as a specific floating point variable is an instance of a float class). Objects generally have attributes and methods. Attributes are variables that are associated with a given object. For example, if \( p \) were an instance (object) of a point class, then \( p.x \) and \( p.y \) would refer to the \( x \) and \( y \) attributes of the object (presumably representing the point’s position) in a plane. Objects can have any number of attributes (including none). Wise users will avoid using those starting with an underscore, as those are, by convention, considered "private" attributes in Python.

Classes can be derived ("inherited") from other classes to give them more specialized behavior. For example, a circle class could be inherited from a geometricShape class. This is also referred to as subclassing. The instances of subclassed objects are considered instances of the parent class as well.

A method, in its most basic form, is simply a function associated with a class, only their calling syntax is a bit different. Taking the point example, a point object may have a translate method which will translate its coordinates by a specified amount. For example, \( p \text{.translate}(1., -4.5) \) would add 1. to \( p.x \) and subtract 4.5 from \( p.y \). The
difference from a more traditional function is that the object and its attributes are naturally associated with the function and are not passed as arguments. Generally any method of a given class will be a method of those that are subclassed from that class as well.

That’s really all you need to know to use Python objects!

---

### 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/stsci/pyssg/lib`, then the syntax would be:

```
% setenv PYTHONPATH /usr/stsci/pyssg/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.

```python
>>> import pyfits
```

#### Convention for Usage Examples

The descriptions in this section include an example of the usage of the PyFITS methods. These examples will all be based on the example set up in “A First Session...” on page 2 and “Working with Tables” on page 5 for images and table examples, respectively. This included the definitions:

- `fimg = pyfits.open('sample1.fits')`
- `prihdr = fimg[0].header`
- `scihdr = fimg[1].header`
- `hdr: any header object, e.g. prihdr or scihdr`
- `scidata = fimg[1].data`
PyFITS Objects and Methods

A FITS file's content is treated by PyFITS as a single Python object. Its methods are responsible for overall I/O for the FITS file and for any new files that would be created from it. The following sections describe the class-level methods and how to access the objects from FITS file.

Header/Data Units

A FITS file consists of one or more Header/Data Units (HDUs). A simple FITS file will only consist of one such Header/Data Unit. Others may have hundreds of HDUs. The FITS standard treats the first of such HDUs a bit differently than the rest and calls it the Primary HDU. Any subsequent HDU is called an Extension HDU. In PyFITS, a FITS file is opened as a 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 extension HDU (or Primary HDU if there is no extension HDU).

Accessing HDUs by Name

- **Syntax**: `fimg[extname, extver]`

This method can be used to access an HDU by the value of the keyword EXTNAME.
- **Example**: `fimg[‘PRIMARY’]`

The above example shows how to access the primary HDU by name.
- **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. See the next example.
- **Example**: `fimg[‘sci’, 2]`

```python
intab = pyfits.open(‘samp_tab.fits’)
tabdat = intab[1].data
tabcols = intab[1].columns
```
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[1:6:3]`

HDULists behave like a Python list, so a user can slice, replace, insert, or append HDU objects in a manner similar to manipulating 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 result in 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**

<table>
<thead>
<tr>
<th>Method name</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>open(filename)</td>
<td>open a FITS file <code>filename</code>, and return a HDUList object</td>
</tr>
<tr>
<td>HDUList()</td>
<td>instantiate (start/create) an empty HDUList</td>
</tr>
<tr>
<td>info()</td>
<td>Prints out a summary of contents for a FITS file</td>
</tr>
<tr>
<td>append(hdu)</td>
<td>add a header data unit <code>hdu</code> as an extension to the HDUList</td>
</tr>
<tr>
<td>flush()</td>
<td>update the associated FITS file on disk with the current version of HDUList in memory</td>
</tr>
<tr>
<td>writeto(new_file)</td>
<td>write the HDUList object in memory to a file named <code>new_file</code></td>
</tr>
<tr>
<td>close()</td>
<td>close the file associated with the HDUList object</td>
</tr>
</tbody>
</table>

**Opening a FITS file**

- **Syntax:** `fimg = pyfits.open(filename, mode="readonly")`
An HDUList is instantiated (or created) from an existing FITS file by using this PyFITS library function. There are three different file I/O modes. The differences reflect what operations are allowed in modifying the file after it is opened. The three modes are:
- **readonly** (default) - do not allow the file on disk to be modified
- **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 existed on file when opened.

Also see “FITS objects and I/O” on page 14 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 FITS 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 `fimg` in memory.

**Writing out a New FITS file**

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

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

**Closing a FITS object**

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

  This method will, if the file was opened in update or append mode, apply any changes of the HDUList to the file on disk by calling the 'flush' method, then close the file on disk.
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 may be modified in memory (even if the file was opened read-only).

No write mode

Observant readers would 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 or update modes (currently, update mode is not supported yet). Data opened in readonly mode may be modified, however the modifications will not appear in the file (the corresponding memory mapped mode is "copy on write"). 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 guaranteed not to appear in the file until flushed or closed. As with regular I/O, if the memory-mapped file is opened in update mode and its size (in FITS blocks) of the header or data unit changes, the elements that change in size will be copied to memory, and when the HDUList is closed or flushed, the memory mapped file will be overwritten to a new file with the resized units.

Creating a new FITS file from scratch

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

  Most FITS files will have their origin in another input file. That is to say, 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 as needed, either by appending HDUs copied from existing FITS files or by creating HDUs from scratch. The first HDU added, though, should be a PrimaryHDU object in order to generate a valid FITS file.

Example 2.1: Example

```python
fitsobj = pyfits.HDUList()
# create Primary HDU
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
fitsobj.writeto('myzeros.fits')

FITS file validity

- Syntax: hdulist.verify()

HDULists do not check for self consistency when changes are made to the HDUList, the HDUs, headers, or data, at the time changes are made. It is possible to construct FITS objects in memory 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 done when the data are written to files or when the verify() method is called for HDUs. For example, calls to verify(), flush(), writeto(), or close() will result in the HDUs being examined for consistency with the FITS standard. What action is taken when inconsistencies are discovered depend on how the HDUList was created. There are several modes that an HDUList can be created with that control how inconsistencies are handled.

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 indicated by NAXIS’s, change the NAXIS’s. The checking includes NAXIS, NAXISi, and BITPIX. Similarly, on output, EXTEND in the Primary HDU is always fixed, regardless of mode. NEXTEND is not modified and should not be considered valid (in general with any FITS package!). The verification modes are not case sensitive.

fix (default)

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 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.

Method of Correction
Only headers are "fixed" when inconsistencies are detected (and the mode is either fix or silentFix). Data are never changed during verification. Fixes to headers are made on the HDUList in memory, unless the file was opened with memory mapping.

Reading non-compliant FITS files
Since PyFITS is relatively new, it has not been tested with a wide variety of FITS data. 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 such FITS files and we will address such cases. It is our general philosophy to correct such non-compliance when reading the data into memory so it will be difficult to write data incompatible with the standard. If this proves to be a serious problem we will reconsider this policy.

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 'image cards.' Each card contains 80 ASCII characters. Most of these cards consist of keyword/value pairs, with an optional associated comment. There also COMMENT and HISTORY cards. PyFITS header objects present two means to read and write information from and to headers, one that 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 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 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 not as 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. This list can be manipulated using all the standard Python list methods.

**Determining the existence of a particular keyword**

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

  Searching for the existence of a particular keyword can be performed using the typical dictionary 'has_key()' method. It will return a 1 if the (case insensitive) keyword exists in the header, and a 0 otherwise.

**Accessing a single keyword value**

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

  This method allows the user to print out the value of a single keyword. If the keyword does not exist, a Python exception will be thrown for the user to handle.

PyFITS is case-insensitive towards keyword names.
**Setting a single keyword value**

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

The value of an existing keyword can be changed in this manner. One cannot add new keywords this way, as the keyword name must already exist for indexing.

**Safely returning a keyword value**

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

One may use the get method as an alternative to simple indexing. This method will return the value of the keyword. However, it has the distinct advantage of returning the specified default value if the keyword is not present.

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

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

**Updating or Adding a Header Keyword**

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

This method allows the user to update the value of an existing keyword or add a new 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 when updating 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 example illustrates how to add the 'TEMP2' keyword after the 'TARGET' keyword in the PRIMARY header.

**Deleting a Header Keyword**

- **Syntax:** del hdr[keyword]

In many ways, a PyFITS header can be viewed as a dictionary with the keyword serving as the key. Deleting a keyword from a header also leverages this behavior by allowing Python’s dictionary delete functionality to operate on headers.

- **Example:** del scihdr['texptime']

Delete the TEXP TIME keyword from the science extension header.
Images and PyFITS memory usage

Astronomical FITS files can be quite large. Some will not fit in memory. PyFITS is developed to provide 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 all the 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 I/O mode, only the headers are read into memory. Data is read into memory only if there is an attempt to read it. For memory-mapped data, memory space is needed only when the data is scaled, i.e. BSCALE != 1 or BZERO != 0.

Example 2.2: Reading data into memory.

```python
>>> 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 # Finally, read data into memory
```

HDULists have a `readall()` method which forces all the data to be read into memory. But if one opens a file in readonly mode, and only accessed the data of one extension, only data for that extension is 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.

```python
>>> 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.**

```python
>>> 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 is 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). When BSCALE and BZERO are applied to data when they are read into memory, they 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 recarray (for record arrays). Recarrays provide two different ways of accessing table data, by rows or columns. The advantage of recarrays 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 mentioned later) in the same way it does in a FITS file, that is, as a repeated series of rows.

Recarrays are inherited from the same base class, NDArray, as numarrays, NDArray, and thus can be accessed and manipulated using all the same mechanisms that NDArray may have. In particular, they may be indexed and sliced in exactly the same way (including use of index arrays). But recarrays also provide special methods that yield numarrays for the
numeric columns of a table. These numarrays are not copies of the data in
the table, but rather, they are a numarray view of exactly the same data in
the table. On the other hand, this means the recarray’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

These concepts are most easily illustrated by an example. This example
will be based on the table whose columns and data are given in Table A.1
on page 27. 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

More advanced methods for working with tables are described later in
this chapter.

```python
>>> 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, 4.000469119440762e-16, 'dead'),
(1788.39599609375, 4.8213738773963954e-13, 'good'),
(1787.678955078125, 3.276061244465643e-13, 'good'),
(1786.961059703125, 8.263330883266397e-13, 'good'),
(1786.2430419921875, 2.2820289494879242e-12, 'good'),
(1785.525024410625, 2.860970928795243e-12, 'good'),
(1784.8079833984375, 8.6380837378829955e-13, 'good'),
(1784.0899658203125, 3.313355991481348e-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
  1787.02050748  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 recarrays uses 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:

<table>
<thead>
<tr>
<th>FITS</th>
<th>recarray</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>b</td>
</tr>
<tr>
<td>B</td>
<td>u</td>
</tr>
<tr>
<td>I</td>
<td>s</td>
</tr>
<tr>
<td>E</td>
<td>r</td>
</tr>
<tr>
<td>D</td>
<td>d</td>
</tr>
<tr>
<td>J</td>
<td>i</td>
</tr>
<tr>
<td>A</td>
<td>a</td>
</tr>
</tbody>
</table>

Displaying information about a table

Several methods exist for obtaining information about a table and its content, including the number of rows and column definitions.

**Getting column definitions for a table**

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

The ColDefs object can be accessed using this syntax. This object contains information about the columns as derived from the header, including: name, format, unit, null, bscale, bzero, disp, start, and dim. These attributes contain the values for all the columns as a list for each attribute.

**Getting number of rows in a table**

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

**Accessing elements of a record (row)**

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

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

**Extracting column attributes**

- **Syntax:** `tabcols.info(coldefs_value)`
Column names are in an attribute of the ColDefs object and a list of them can be returned using this method. For example, the sample table has columns named "WAVELENGTH", "FLUX", and "QUALITY". The characteristics for those columns can be accessed, as illustrated in Example 2.5.

**Example 2.5: Accessing column attributes**

```python
>>> 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('format')
format:
  ['1r', '1r', '8a']
>>>  # Or you can use cols.names and colsformats
```

**Additional Column Methods**

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

**Using Chararrays**

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

Since tables may contain character fields, a special array has been constructed to handle string arrays. Unlike Python strings, string arrays contain fields of fixed size. Like NDArrays, they may be indexed and sliced the same way. Generally, individual items are returned as Python strings. 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 chararray.

**Example 2.6: Printing chararray members**

```python
>>> 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, nrows, fill, tbtype)

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 ColDef 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 is supported now)

Note that the arrays used to construct the new table are referred to by a Column list or ColDef object and it is in constructing either one of these that the data is provided for the new table. The arrays so provided are copied to a single data buffer allocated for the new table. Tables, once created, are 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

```python
>>> targets = chararray.array(['M51','NGC4151','Crab Nebula','Beta Pictoris'])
>>> col1 = pyfits.Column(name='targname', format='13a', array=targets)
>>> col2 = pyfits.Column('RA',format='r', unit='hours', array=numarray.array([1.,2.,3.,4.]))
>>> col3 = pyfits.Column('DEC','r','degrees',array=numarray.arange(4))
>>> tabhdu = pyfits.new_table([col1, col2, col3], nrows=100)
```
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

```python
>>> 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 the a new table can be created from an old one with additional rows.

Example 2.9: Adding rows to a table.

```python
>>> 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 is taken directly from the table data buffer. But Boolean columns, columns in an FITS ASCII table, and columns with BSCALE != 1 or BZERO != 0, are copied to new arrays.
## Appendix A: Source For Examples

In The Appendix...

Sample Table / 27

---

### Sample Table

**Table A.1**: Sample Table for Examples

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