A Python Module for FITS Files with Full C Level Programming Functionality

Ian Bond
Institute of Natural and Mathematical Sciences
Massey University
Private Bag 102-904, North Shore Mail Centre
Auckland, New Zealand
i.a.bond@massey.ac.nz

Abstract

A Python module for manipulating files in the FITS format is described. The module was constructed using the capabilites of `ctypes` to dynamically create foreign function interfaces from a C library. Here this was used to import the CFITSIO library into Python. I describe how this module can be used to call the functions from the C library in their near native form, and how one can manipulate FITS files in a style that Python programmers are accustomed. The `ctypes` and `ctypeslib` modules allows one to import all routines and data structures from the C library and avoids the need to manually write language bindings for each routine. Moreover, these modules allow the Python programmer to enjoy the full functionality of the the underlying C library.

Keywords*: flexible image transport system, ctypes, foreign function interface.*

1. Introduction

The Flexible Image Transport System (FITS) is a digital file format that grew out of the needs of the astronomical community (Wells, Greisen, & Hartan 1981). It is the standard for storing, analyzing, and transporting data based on astronomical observations using all telescopes and instruments that operate at wavelengths from radio to gamma rays. The FITS format is primarily used for storing astronomical imaging data, but it is also used for spectroscopic and tabular data. FITS support is available in most specialized astronomical data analysis platforms such as IRAF (Tody 1993), and is also available to a limited extent in some popular off-the-shelf open source and commercial applications such as Gimp and Photoshop. For C programmers, the CFITSIO library (Pence 1999) is available and provides an extensive and well documented library that has everything one needs to read, write, and manipulate FITS files.

If programmers in other languages wish to make use of C library functions, the common practice is to develop "wrapper" functions that utlize the foreign function interface capabilites of the higher level language. Also these wrappers are designed to hide any messy details involved in calling the C functions, behind clean interfaces that also provide a convenient user view of the library. Existing Python wrappers for CFITSIO based on this approach include `pyfits` (Barrett and Bridgeman 1999) and `pfits` (Parsons 2013).

Implementing Python bindings to C libraries can be a tedious affair – in particular, CFITSIO contains over 330 routines. Consequently not all features of the library may be implemented. Also, one is tied into a particular user view of the developer of the Python bindings, and what may be a convenient user view for some programmers may not be the case for others.

In this paper I discuss an approach using the capabilities of the Python `ctypes` and `ctypeslib` modules to automatically generate foreign function interfaces. The power of `ctypes` and `ctypeslib` is being increasingly recognized. For example, Kloss (2011) discusses this approach for the LittleCMS colour management library. With a single import, one has access to the full functionality and features provided by the underlying C library. I will then show how it is straightforward to write one's own custom wrappers that are convenient to each individual user and project. I also show how one can manipulate FITS files in a "pythonic" manner whereby file objects are created and manipulated in a way that Python programmers are accustomed and follows common Python programming style and practice. All these capabilities are conveniently packaged into a Python module I have developed called `fitsio`. I have made this module available through the Google Project Hosting Service under the code license GNU GPL v3. Detailed instructions on how to obtain and install this package are given in Appendix A.

The FITS format is used extensively by astronomers engaged in data analysis. However, scientists and engineers outside that community, who work with imaging data, may also find the FITS format useful. The main purpose of this paper is to provide a how-to guide for Python programmers who wish to use the CFITSIO library through the `ctypes` mechanism. The ideas described in this paper were presented at the 2011 Kiwi Python meeting, and this paper is also an attempt to adapt the related handouts for that presentation into a full written article.

2. Overview of the Python Module

2.1. Anatomy of a FITS file

The FITS standard is maintained by the FITS Working Group of the International Astronomical Union. The full technical specification for the FITS standard can be found at the FITS Support Group[1]. There are also many articles and online resources at varying levels of technical detail that describe the FITS standard and how one can work with files in that format. Here a broad description of the structure of a FITS formatted file is given.

A FITS file comprises one or more header-data units (HDUs). The "header" part of an HDU component is a sequence of 80 character ASCII records that encode metadata for the "data" part. Each record in the header can either comprise a keyword-value-comment triplet or can be a special record identified by either the `COMMENT` or `HISTORY` keywords. The first header-data unit is commonly referred to as the primary HDU. Any subsequent HDUs are known as FITS extensions. A FITS file with more than one HDU is often referred to as a multi-extension FITS file. In many cases it is sufficient to use FITS files without any extensions – for example storing a grayscale two dimensional image from a CCD camera. In

[1] See `http://fits.gsfc.nasa.gov/`

fact, there are a number of applications that claim support for FITS but do not support multi-extension FITS files.

The data component of an HDU commonly comprises an array of values in binary format that correspond to the values in each pixel of a two dimensional image. However, ”imaging” arrays of any number of dimensions are possible. In the primary HDU, the data may only be imaging data (or it could be empty). However, for HDUs that are part of the extensions, the data component can be used to store imaging data or alternatively store tabular data in either ASCII or binary format. For tabular data the associated header records are used to encode metadata for the fields of the table. In both imaging and tabular data, all manner of word sizes (`short`, `int`, `float`, etc) are allowed.

2.2. Importing CFITSIO into Python

The CFITSIO library provides a full suite of routines for manipulating and navigating the components of a FITS file. Common operations include opening and closing files, creating new files, accessing header records by keyword, reading/writing imaging and tabular data, plus many other operations. The reader is refered to the CFITSIO reference manual for more information and the full documentation of all the routines in the library (Pence 2004).

The build process can be configured to produce a shared library file. The Python `ctypes` foreign function interface can then be used to load all routines in the library without having to explicitly define bindings for them. This is done by a call to `ctypes.loadDLL()`, with the details depending upon the OS platform. On a Linux platform, for example, one may use something like:

```
import ctypes
cfitsfio = ctypes.loadDLL('/usr/local/lib/libcfitsio.so')
```

With this call, all routines in the CFITSIO shared library are available in the Python script and can be accessed through the above `cfitsio` namespace. There are, however, some other steps that need to be carried out before using the routines.

Each `struct` datatype in the C library needs to be mapped to a Python class that extends `ctypes.Structure`. Each field in the C structure requires a counterpart in the python class that associates a datatype class instance from `ctypes` with the C datatype. Also it is desirable to specify the types of the parameters and return types for each function. If done manually, this can be tedious and error prone for libraries such as CFITSIO that define large numbers of routines and uses elaborate C structures. In particular the `fitsptr` structure contains several nested structures that go several layers deep, and some of these sub-structures contain a large number of fields.

Fortunately the Python `ctypeslib` module can be used to automatically generate python code from the structure and function definitions provided in C header files. This module will also map out any symbolic constants defined in the C header file via the `#define` construct. The code generation is a two step process. First `gccxml` is used to convert the definitions in the library header file (in this case `fitsio.h`) into an XML file. This XML file is then

sourced in the second step to generate python code that maps all structures and routines. I have written a script, `generate_stubs.py` to carry out these steps in one smooth operation. Each routine in the CFITSIO library is defined by a short name together with a longer alias. The `ctypeslib` module was able to resolve all function aliases, but there were some "unresolved aliases" associated with some constants. However, I have not yet encountered any problems.

2.3. The fitsio Moodule

I provide here a python module called `fitsio` that contains two sub-modules: `cfitsio` and `pyfitsio`. The Python code generated in the above procedure are all gathered into `cfitsio`. All CFITSIO routines, data structures, and symbolic constants can be accessed in a Python script via an import such as:

```
from fitsio import cfitsio
```

This simple import is all that is needed to handle FITS files with access to the full functionality offered by CFITSIO. In Section 3, I describe how one can manipulate FITS files using just the C routines that are imported via `cfitsio`. In Section 4, I show how one can use the `pyfitsio` module to manipulate FITS files in a pythonic manner. This module provides a class implementation to represent FITS files together with an exception class.

The examples given here are not exhaustive and do not cover all of the capabilities of CFITSIO. However, the CFITSIO library is well documented and the abilty of `ctypes` to dynamically generate a foreign function interface from the shared library gives the Python programmer the same level of access to CFITSIO as a C programmer. These examples will show that the Python programmer can select any of the CFITSIO routines and quickly write their own convenience functions to suit their own particular style and needs. The emphasis here is on *how* rather than *what*.

3. Python CFITSIO Interface

3.1. Opening and closing FITS files

The CFITSIO routine to open an existing FITS file is defined in the C header file as:

```
int ffopen(fitsfile** fptr, char* filename, int iomode, int* status);
```

The `fitsfile` construct defines a C structure in `fitsio.h` which is mapped to a Python class called `fitsfile` in the code generated by `ctypeslib`. Most of the routines in CFITSIO are defined using a very similar pattern. The first argument is a pointer or double pointer to `fitsfile` and the last argument is a pointer to a status variable in which a non-zero value means that some problem had occurred in the call. Also a non-zero value is returned by the function in case of any problem.

In opening a FITS file, the input/output mode is specified by assigning one of the symbolic constants, `READONLY` or `READWRITE`, defined in `fitsio.h` to the variable `iomode`. These

constants are mapped into the Python code for the `cfitsio` module that is generated by `ctypeslib`. The following example shows how one could write a convenience function in Python to open an existing FITS file as read-only:

```
import ctypes
from fitsio import cfitsio
def open_fits(filename):
    # Get a pointer to the FITS file
    fptr = ctypes.pointer(cfitsio.fitsfile())
    # Initialize status to zero
    status = ctypes.c_int(0)
    # Open the file. Note the use of byref to get the value of the pointer
    # to fitsfile()
    cfitsio.ffopen(ctypes.byref(fptr), filename, cfitsio.READONLY,
                   ctypes.byref(status))
    return fptr
```

To close a FITS file, the following CFITSIO routine is available

```
int ffclos(fitsfile* fptr, int* status);
```

In Python this can be implemented simply as

```
def close_fits(fptr):
    status = ctypes.c_int(0)
    cfitsio.ffclos(fptr, ctypes.byref(status))
```

A non-zero value for the status variable means that some problem had occurred in the call to the CFITSIO routine. For example, if one attempts to open a nonexistent FITS file, the status will be set to the value defined by the symbolic constant `FILE_NOT_OPENED` in the CFITSIO header file. CFITSIO provides a number of routines for generating error messages. One of these is

```
int ffgerr(int status, char* err_text);
```

In Python, one could then write an error handler, that uses this routine, as follows:

```
import sys
import ctypes
from fitsio import cfitsio
def handle_error(status):
    '''Print out CFITSIO error message and exit program.'''
    # Create a string buffer to hold the error message
    errtext = ctypes.create_string_buffer('\000' * cfitsio.FLEN_STATUS)
    cfitsio.ffgerr(status)
    print 'Error status code =', status.value
    print 'Error message reads:', errtext.value
    print 'Exiting program'
    sys.exit(status.value)
```

Note here that `status` is not a primitive `int`, but a class instance generated by the `ctypes.c_int()` routine.

I re-emphasize here that the design of convenience functions such as these is purely a matter of choice for the individual programmer. The users have considerable flexibility in designing an error handler that best suits their needs. In Section 4 I will describe an exception class to handle non-zero status values.

3.2. Navigating the HDUs

As mentioned earlier, a FITS file can have one or more header-data units. The following CFITSIO routine can be used to find out the number of HDUs in a given FITS file:

```
int ffthdu(fitsfile* fptr, int* hdunum, int* status);
```

When the file is first opened, the file pointer is located at the primary HDU. One can move to any numbered HDU using the following CFITSIO routine

```
int ffmahd(fitsfile* fptr, int hdunum, int* hdutype, int* status);
```

The data in an HDU can either be imaging data, an ASCII table, or a binary table. These correspond to the symbolic constants `IMAGE_HDU`, `ASCII_TBL`, and `BINARY_TBL`. The following Python code shows how these routines can be used to walk through the HDUs in an opened FITS file and examine their types.

```
def check_hdus(fptr):
    status = ctypes.c_int(0)
    # Get the total number of HDUs
    hdunum = ctypes.c_int()
    cfitsio.ffthdu(fptr, ctypes.byref(hdunum), ctypes.byref(status))

    # Loop over all HDUs and compare their type with possible values
    for n in range(1, hdunum.value+1):
        hdutype = ctypes.c_int()
        cfitsio.ffmahd(fptr, n, ctypes.byref(hdutype),
                       ctypes.byref(status))
        if hdutype.value == cfitsio.IMAGE_HDU:
            hdu_descr = 'Image HDU'
        elif hdutype.value == cfitsio.ASCII_TBL:
            hdu_descr = 'ASCII table'
        elif hdutype.value == cfitsio.BINARY_TBL:
            hdu_descr = 'Binary table'
        else:
            hdu_descr = 'Unknown HDU type'
        print n, hdutype.value, hdu_descr
    # Move back to the primary HDU
    cfitsio.ffmahd(fptr, 1, ctypes.byref(hdutype), ctypes.byref(status))
```

Note that the numbering of HDUs starts at 1 for the primary HDU.

3.3. Reading and writing FITS headers

The CFITSIO library provides an extensive set of routines for manipulating the header component of HDUs in a FITS file. The most common operations are reading and writing a

value associated with a keyword name. However, routines are available for other operations like stepping sequentially through the header records and copying headers from one opened FITS file to another. The reader is referred to the CFITSIO reference manual for more details.

One CFITSIO routine that can be used to write a new keyword-value-comment record is

```
int ffpky(fitsfile* fptr, int datatype, char* keyname, DTYPE* value,
          char* comment, int* status);
```

This writes the value and comment string associated with the given key name. Here `DTYPE` can be any of the primitive data types (`short`, `float`, etc). The parameter `datatype` tells FITSIO the data type that will be used in the C program – this value must be set to one of the symbolic constants defined in `fitsio.h` (refer the reference manual). Where possible, CFITSIO will perform automatic type conversion from `DTYPE` regardless of how the value is encoded in the header.

A Python implementation of this function whereby the value is encoded as a `float` could be:

```
def write_key_as_float(fptr, keyname):
    status = ctypes.c_int()
    value  = ctypes.c_double()
    comment = ctypes.create_string_buffer('\000' * cfitsio.FLEN_COMMENT)
    cfitsio.ffpky(fptr, cfitsio.TDOUBLE, keyname, ctypes.byref(value),
                  comment, ctypes.byref(status))
```

A CFITSIO routine to read a value and comment for a given key name is:

```
int ffgky(fitsfile* fptr, int datatype, char* keyname, DTYPE* value,
          char* comment, int* status);
```

In a very similar fashion, one could write a Python function to implement this as follows:

```
def read_key_as_float(fptr, keyname):
    status = ctypes.c_int()
    value  = ctypes.c_double()
    comment = ctypes.create_string_buffer('\000' * cfitsio.FLEN_COMMENT)
    cfitsio.ffgky(fptr, cfitsio.TDOUBLE, keyname, ctypes.byref(value),
                  comment, ctypes.byref(status))
    return value.value
```

In this example, the comment string is discarded and only the value associated with the key is returned.

3.4. Reading and writing imaging data

As with writing and reading FITS headers, CFITSIO provides a number of routines for writing and reading pixel data to and from the data component of the current HDU. Before writing imaging data to a FITS file, it is necessary to set up the header with the appropriate

records giving the image dimensions and the imaging data encoding as the number of bits per pixel. This can be done with the following CFITSIO routine:

```
int ffccrim(fitsfile* fptr, int bitpix, int naxis, long* naxes,
            int* status);
```

Here `naxis` is the number of dimensions of the image with the sizes in each dimension stored in the array pointed to by `naxes`. This is straighforward to implement in Python. Suppose, for example, one wishes to write a two dimensional image with 256 columns and 301 rows with 32 bit floats per pixel. The above routine would then be called as follows:

```
npixx = 256
npixy = 301
# Make a ctypes array of the sizes in each dimension
naxes = (ctypes.c_long * 2)(npixx, npixy)
cfitsio.ffcrim(fptr, cfitsio.FLOAT_IMG, 2, naxes, ctypes.byref(status)
```

After calling `ffcrim`, the FITS file can accept imaging data to be written. A CFITSIO routine to write pixel data to a FITS file is:

```
int ffppx(fitsfile* fptr, int datatye, long* fpixel, long nelements,
          int naxis, long* naxes, int* status);
```

This will write a number of pixels specified by `nelements` onto a FITS file with the coordinates of the first pixel on the file given by the array `fpixel`. The array of pixels is addressed by `DTYPE* array` which must point to enough memory. If `fpixel` is set to the first pixel in the file and `nelements` is set to the total size of the array, then the entire image will be written in a one-to-one correspondence between memory and file. This routine will work for any number of dimensions. Also the variable `datatype` and `DTYPE` function in the same way as automatic data conversion mechanism for reading from headers.

This routine provides a lot of functionality and would present a number of dilemas if one was to try and hand code a C binding to this method using, for example, the native Python API. Some Python users may wish to write the data from a C style array which can be referenced in `ctypes`. Others may wish to write from a multi-dimensional `numpy` array. Some users may wish to have the capability of writing sub rasters of the imaging data, whereas it may suit others to dump the entire image into the file in one go. Moreover, there is the choice of data type to consider. All of this presents a number of design decisions faced by the developer of a C binding.

However, the ability to dynamically load the CFITSIO library into Python, makes it straightforward for individual users to quickly implement a wrapper to this function that is best suited to their specific needs. For example, consider a project where one is dealing exclusively with 2 dimensional imaging data stored in `numpy` arrays with 32 bit floating type per element of the array (`dtype=numpy.float32`). A Python function such as the followng could serve as a useful convenience function that writes the entire array onto the current HDU that is pointed to by the `cfitsio.fitsfile()` instance `fptr`

```
def write_numpy_2d(fptr, pixdata):
    # Number of X (columns) and Y (rows). Note the row major format
    npixy, npixx = pixdata.shape
    # C style pointer to the numpy array
    pixptr = pixdata.ctypes.data_as(ctypes.POINTER(ctypes.c_float))
    fpixel = (ctypes.c_longlong * 2)(1,1)
    nelements = ctypes.c_longlong(npixx * npixy)
    status = ctypes.c_int(0)
    cfitsio.ffppx(fptr, TFLOAT, fpixel, nelements, dataptr,
                  ctypes.byref(status))
```

Note here the need to obtain a C style pointer to the `numpy` array that is then used as input to `ffppx`. The above example shows how this is done. Also, the data type specified in the 2nd argument of the CFITSIO routine must match the data type of the `numpy` array.

Now suppose one wish to read the pixel data in the FITS array into two dimensional `numpy` array. First one could use the following CFITSIO routine that determines the image dimensions and pixel encodings from the header:

```
int ffgpxv(fitsfile* fptr, int datatype, long firstelem, long nelements,
           DTYPE* nulval, DTYPE* array, int* anynul, int* status);
```

This is straightforward to call in Python:

```
maxdim = 2
naxes = (ctypes.c_longlong * maxdim)()
naxis = ctypes.c_int()
bitpix = ctypes.c_int()
status = ctypes.c_int(0)
cfitsio.ffgpxv(fptr, maxdim, ctypes.byref(bitpix), ctypes.byref(naxis),
               naxes, ctypes.byref(status))
npixx, npixy = naxes
```

If a two dimensional image is expected, a check could be added to the above code comparing the image dimensions in the FITS header, `naxis`, with `maxdim`.

The following CFITSIO routine reads imaging data from a FITS file into memory with starting address `DTYPE* array`

```
int ffgpxv(fitsfile* fptr, int datatype, long firstelem, long nelements,
           DTYPE* nulval, DTYPE* array, int* anynul, int* status);
```

This also allows one to check for undefined pixels in the FITS array where the pixels values are equal to `nullval`, with the total number of such pixels return in `anynul`. If `nulval=0`, no checks are made. Suppose one simply wished to gobble the entire array into a two dimensional `numpy` array. Given the image dimensions determined from above, here is a possible Python implementation:

```
imdata = numpy.zeros( (npixy, npixx) )
dataptr = imdata.ctypes.data_as(ctypes.POINTER(ctypes.c_double))
fpixel = (ctypes.c_longlong * 2)(1,1)
```

```
nelements = ctypes.c_longlong(npixx * npixy)
nullval = ctypes.c_double(0.0)
anynull = ctypes.c_int()
status = ctypes.c_int(0)
cfitsio.ffgpxv(fptr, TDOUBLE, fpixel, nelements, ctypes.byref(nullval),
                   dataptr, ctypes.byref(anynull), ctypes.byref(status))
```

These code fragments can easily be adapted to suit the needs of the individual user. For example, some may wish to store the data as a flat one dimensional array, regardless of the number of dimensions of the image represented in the FITS file. Others may prefer not to use `numpy` arrays at all, and just store the data in `ctypes` arrays.

In the code samples that come with the `fitsio` package, I provide examples of generalized image read and write routines that will work for any number of dimensions and will match the dimensions in the FITS arrays with those in the `numpy` arrays.

3.5. Reading and writing tabular data

As mentioned earlier, the data component of a primary HDU can only be imaging in nature. However, for HDUs that are part of the FITS extensions, the data can be tabular rather than imaging. The CFITSIO library provides a number of routines for manipulating and reading FITS tables, and reading table metadata. All of these are well documented in the CFITSIO Reference Guide and can be easily adapted to Python.

To create a table in the current HDU, the following CFITSIO routine can be used:

```
int ffcrtbl(fitsfile* fptr, int tbltype, long naxis2, int tfields,
            char* ttype[], char* tform[], char* tunit[], char* extname,
            int* status)
```

The type of table (ASCII or binary) is specified by `tbltype` and is set to one of the symbolic constants `ASCII_TBL` or `BINARY_TBL`. The number of rows is specified by `naxis2` while `tfields` gives the number of columns. The names of each column are given by `ttype[]`, while `tform[]` and `tunit[]` list the printing formats and physical units of the columns.

When using this routine in Python the tricky part here is in implementing the arrays of strings in the above routine. The easiest way to do this is to use `ctypes` arrays. For example, for a table with 3 columns named `Planet`, `Diameter`, and `Density`, an array suitable for input into `ffcrtbl` can be constructed as follows:

```
ttype = (ctypes.c_char_p * 3)('Planet', 'Diameter', 'Density')
```

Their formats and units can also be set up in a similar fashion:

```
tform = (ctypes.c_char_p * 3)('a8', 'I6', 'F4.2')
tunit = (ctypes.c_char_p * 3)('\0', 'km', 'g/cm^3')
```

An ASCII table in the current HDU can then be created as follows:

```
tfields = ctypes.c_int(3)
nrows   = ctypes.c_longlong(6)
extname = 'PLANETS_ASCII'
cfitsio.ffcrtbl(fptr, cfitsio.ASCII_TBL, nrows, tfields, ttype, tform,
                tunit, extname, ctypes.byref(status))
```

Having created a table in the current HDU, data can then be written to the table column by column using the following CFITSIO routine:

```
int ffpcl(fitsfile* fptr, int datatype, int colnum, long firstrow,
          long firstelem, long nelements, DTYPE *array, int* status)
```

Here the column number is specified along with a pointer to the array with the data for the column. The following code fragment shows how this could be called in Python:

```
planet = (ctypes.c_char_p * nrows)('Mercury', 'Venus', 'Earth', 'Mars',
                                   'Jupiter', 'Saturn')
diameter = (ctypes.c_long * nrows)(4880, 12112, 12742, 6800, 143000,
                                   121000)
density  = (ctypes.c_float * nrows)(5.1, 5.3, 5.52, 3.94, 1.33, 0.69)
firstrow  = ctypes.c_longlong(1)
firstelem = ctypes.c_longlong(1)
cfitsio.ffpcl(fptr, cfitsio.TSTRING, 1, firstrow, firstelem, nrows,
              planet, ctypes.byref(status))
cfitsio.ffpcl(fptr, cfitsio.TLONG, 1, firstrow, firstelem, nrows,
              diameter, ctypes.byref(status))
cfitsio.ffpcl(fptr, cfitsio.TFLOAT, 1, firstrow, firstelem, nrows,
              density, ctypes.byref(status))
```

The above can be easily adapted if the column data were encoded in `numpy` arrays. One just has to make sure that a pointer to the array is obtained, as described in the previous sub-section, and use that as the input to `ffpcl`.

Reading data from a table can also be done out column by column using the following CFITSIO routine:

```
int ffgcv(fitsfile* fptr, int datatype, int colnum, long firstrow,
          long firstelem, long nelements, DTYPE* nulval, DTYPE *array,
          int *anynul, int* status)
```

To use this in Python to read, for example, the density column where a value of -99.0 is defined as a null value, one can can use something like the following;

```
frow = ctypes.c_longlong(1)
felem = ctypes.c_longlong(1)
nullval = ctypes.c_float(-99.)
density = (ctypes.c_float * nrows)()
anynulls = ctypes.c_int()
cfitsio.ffgcv(infptr, cfitsio.TFLOAT, colnum, frow, felem, nrows,
              ctypes.byref(nullval), density, ctypes.byref(anynulls),
              ctypes.byref(status))
```

Again, it can be easily adapted if one prefered to read the data into a `numpy` array instead of a `ctypes` array.

4. Python Style Access to FITS Files

The routines provided in the Python `cfitsio` module provide the full functionality of the CFITSIO library at the same level of access to a C programmer. However, it would be desirable to be able to manipulate FITS files in a pythonic manner similar to how one may work with other types of files. To this end, I have included a sub-module `pyfitsio` within `fitsio`. This provides a Python class, `FitsFile`, to represent FITS files, together with a factory style `open()` function that can be used to create instances of that class. Also provided is an exception class to represent non-zero status values resulting from any calls to the CFITSIO routines.

The `FitsFile` class provides a number of routines that function as wrappers to some of the simpler and more commonly used CFITSIO routines for navigating and manipulating the HDUs. However, the design of the class has been intentionally kept short and simple and the more complex CFITSIO routines have been left out. This is for the reasons discussed in the previous section where it is not very practical to try and design a wrapper for every one of these routines.

The crucial data member of the `FitsFile` class is a `ctypes` pointer defined as follows:

```
class FitsFile(object):
    def __init__(self):
        '''Initialize with an empty C fitsfile pointer'''
        self.fptr = ctypes.pointer( cfitsio.fitsfile() )
```

This pointer can be used as input to any of the relevant `cfitsio` routine as best illustrated by the following walkthrough.

First import both the modules:

```
from fitsio import cfitsio, pyfitsio
```

Create a new FITS file instance for writing. If the “clobber” flag is set, then any existing file with that name will be overwritten.

```
ff = pyfitsio.open(filename, mode='w', clobber=True)
```

Now suppose one has a `numpy` array of data, `pixdata`, encoded as 16 bit integers per element (`dtype=int16`), and we wish to write this to the FITS file as 32 bit floats per pixel. First one sets up the FITS header with the dimensions of the array. Note the need to reverse the array of dimensions because `numpy` stores its data in row major ordering whereas CFITSIO uses column major ordering.

```
naxes = list(pixdata.shape)
naxes.reverse()
ff.create_image(naxes, cfitsio.FLOAT_IMG)
```

To write the pixel data to the image, we here make use of the following CFITSIO routine

```
int fits_write_image(fitsfile* fptr, int datatype, long firstelem,
                     long nelements, DTYPE* array, int* status)
```

This is used as an alternative to the routine used in the previous section because here all one needs is a single pointer, rather than an array, to the start position in the FITS file and the total number of elements to write. Note also that here I am using the alias of the C function that named `ffppr`. If we simply wish to write the entire `numpy` array to the start of the data component of the HDU, one can come up with a convenience function such as the following:

```
def numpy_to_fits(pixdata, ff):
    # Get a C pointer to the array
    pixptr = pixdata.ctypes.data_as(ctypes.POINTER(ctypes.c_short))
    npix = pixdata.size
    status = ctypes.c_int(0)
    fpixel = ctypes.c_longlong(1)
    nelements = ctypes.c_longlong(npix)
    cfitsio.fits_write_img(ff.fptr, cfitsio.TSHORT, fpixel, nelements,
         pixptr, types.byref(status))
```

Note that in this case, the data type parameter, `cfitsio.TSHORT`, must match the type of the `numpy` array. CFITSIO then performs automatic type conversion to the 32 bit float pixel data encoding. Again, if one was using different data structures to hold the pixel data, then it is straightforward to come up with similar convenience functions. In the case of complex CFITSIO routines such as the above, it does not make much sense to try and come up with a wrapper that suits everyone's requirements, rather it is better to make it easier to use an instance of the `FitsFile` class with the CFITSIO routines.

The `FitsFile` class provides some functions to write and read header records. So in this example we add a few keyword-value-comment records together with a `COMMENT` record:

```
ff.write_key('EXPTIME', 302.2, 'Exposure time in seconds')
ff.write_key('SERIALNO', 12345, 'Serial number of this file')
ff.write_key('NAME', 'Ian Bond', 'The creator of this file')
ff.write_comment('This file is the result of a demo program')
```

When we are finished with the FITS file, it can be closed in a pythonic manner

```
ff.close()
```

In continuing with Python style object-oriented programming, an exception mechanism would be desirable in case things go wrong. As mentioned earlier, a non-zero value of the `status` variable from a call to a CFITSIO routine indicates some problem had occurred. These include attempting to open a non-existent file for reading, attempting to access non-existent keywords in the header, and many more. The CFITSIO library defines 139 error

status codes as symbolic constants in `cfitsio.h`. The library also provides some routines to generate human readable message strings associated with the integer status values.

The `pyfitsio` module provides an exception class, `FitsIOError`, that is raised by all functions in the `FitsFile` class in case of a non-zero status value. Also provided is a function `check_error` that will generate and raise the `FitsIOError` exception if that status value input is non-zero. This can be used, for example, in the above convenience function by simply adding the following line at the end. Note if the status value is zero, ie no error, the routine does nothing

```
pyfitsio.check_error(status)
```

The following code fragment shows how the exception class can be used. In this example, a FITS file is input for reading and the header is examined for particular keywords:

```
try:
    ff = pyfitsio.open(filename, mode='r')
    naxes = ff.read_axes()
    expo = ff.read_key('EXPTIME')
    ff.close()
    print naxes, expo
except pyfitsio.FitsIOError, e:
    print 'Caught a problem with the FITS file'
    print str(e)
except Exception, e:
    print 'Got some other problem'
    print str(e)
```

Note here that the `FitsFile` class only implements a small subset of the CFITSIO routines for manipulating individual FITS files. No attempt has been made to try and implement a pythonic wrapper to all of the CFITSIO routines. This would be overkill, and would defeat the purpose here of being able to use `ctypes` to automatically generate a foreign function interface to a C library. It is intended here that the `pyfitsio` module be used in conjunction with `cfitsio`, where instances of `FitsFile` are used for relatively simple operations such as opening and closing files and the user writes their own custom wrappers to the `cfitsio` routines to perform more comples tasks such as reading and writing imaging and tabular data.

5. Conclusion

CFITSIO is the most extensive library for C programmers for manipulating FITS files. The routines in this library can be easily imported into Python as a foreign function interface using `ctypes`. Also `ctypeslib` can automatically generate Python code that mirrors any C structures and symbolic constants defined in the C header file. I argue that directly importing a C library in this manner is a preferable approach than going through the tedious process of developing Python bindings. This is particularly true if there are many complex routines in the libary. As long as the C library is well documented, as is the case for CFITSIO, and the programmer is competent with `ctypes`. It is then straightforward for the programmer to simply choose the C routine that is required, and then start using it in Python.

I have developed a Python module to make it easy to import the CFITSIO library and to handle FITS files in a manner consistent with a pythonic style of programming. All of this immediately provides a Python programmer the same level of access and functionality of the library as a C programmer.

Acknowledgements

I am grateful to Guy Kloss for introducing me to `ctypes` and `ctypeslib`, and for reading over this manuscript helping me to improve the final version.

References

Barrett, P.E. And Bridgeman, W.T. (1999): *PyFits, a FITS Module for Python*, in ASP Con. Ser. Vol. 172, Astronomical Data Analysis Software and Systems, eds. D. M. Mehringer, R. L. Plante, & D. A. Roberts, p. 483

Kloss, G. (2013): `https://launchpad.net/pylittlecms/littlecms-0.5.1`, last retrieved 2013 March 10

Parsons, A. (2013): *A Python FITS interface built using CFITSIO*, `https://pypi.python.org/pypi/pfits`, last retrieved 2013 March 10

Pence, W. (1999): *CFITSIO* v2.0, in ASP Con. Ser. Vol. 172, Astronomical Data Analysis Software and Systems, eds. D. M. Mehringer, R. L. Plante, & D. A. Roberts, p. 487

Pence, W. (2004): *CFITSIO User's Reference Guide: an Interface to FITS Format Files for C Programmers*, available from NASA's High Energy Astrophysics Science Archive at `http://heasarc.gsfc.nasa.gov/fitsio/`, last retrieved 2013 July 18.

Tody, D. (1993): *IRAF in the Nineties*, in ASP Con. Ser. Vol. 52, Astronomical Data Analysis Software and Systems, eds. R.J. Hanisch, R.J.V. Brissenden, and J. Barnes, p. 173

Wells, D.C., Greisen, E.W., and Harten, R.H. (1981): *FITS: A Flexible Image Transport System*, Astronomy and Astrophysics Supplement Series, 44, 363

Appendix A: Installation

The following installation instructions are for Linux systems (on which `fitsio` was developed and tested). They can be adapted to other platforms.

First download the CFITSIO library available from NASA at

```
http://heasarc.gsfc.nasa.gov/fitsio/
```

Follow the installation instructions given with the package, making sure that a shared or dynamic library is built:

```
make shared
```

The Python `fitsio` modules are hosted through the Google Project Hosting service. The package can be fetched using Mercurial as follows:

```
hg clone https://ian.b.007@code.google.com/p/pyfitsio/
```

Unpack the distribution and change to the package top level directory. Examine the script `generate_stubs.py` and modify the constants specifying the paths to the CFITSIO header file and library file. Then run the script:

```
./generate_stubs.py
```

there may be some "unresolved aliases" errors but these shuld not be a problem. This script will produce the file `_cfitsio.py` in the `fitsio` sub-directory. Do not edit this file as it is automatically generated. This file contains all function definitions, data structures, and symbolic constants that were found in the CFITSIO header file.

The modules are now ready to use. You can either just add the current path to `fitsio` to your `$PYTHONPATH` environment or install in the system Python module path.

Go the samples directory and try running the example scripts that are provided.

Appendix B: List of sample scripts

CFITSIO cookbook

The CFITSIO distribution includes a sample program `cookbook.c` which provides a number of routines showing how to read and write imaging and tabular data to and from FITS files, plus a routine to show how headers can be copied from one file to another. This Python module includes a sample script `cookbook.py` which shows how Python programmers can use the C routines in `cookbook.c`. This sample program does not use the pythonic module. It just imports the `cfitsio` module and uses the imported routines in (almost) the same way as they are done in the C program.

Python FITS file access

A series of sample scripts are provided that implement each of the routines in the above cookbook in a pythonic manner using both the `cfitsio` and `pyfitsio` module. Each routine is implemented as a separate script:

`writeimage.py`

`writeascii.py`

`writebintable.py`

`copyhdu.py`

`selectrows.py`

`readheader.py`

`readimage.py`

`readtable.py`

These can be run separately or they can all be run one after the other by launching the following script:

`run_demos.py`

An additional example script `readff.py` is also provided that shows how to walk through the HDUs of a multi-extension FITS file.

FITS and numpy arrays

A sample script `fitsnp.py` is provided that shows how to write and read imaging data, of any number of dimensions, that are stored in `numpy` arrays.