Using Python for Interactive Data Analysis

Perry Greenfield
Robert Jedrzejewski
Vicki Laidler
Space Telescope Science Institute

April 4, 2005

1 Purpose

This is intended to show how Python can be used to interactively analyze astronomical data much in the same way IDL can. The approach taken is to illustrate as quickly as possible how one can perform common data analysis tasks. This is not a tutorial on how to program in Python (many good tutorials are available—targeting various levels of programming experience—, either as books or on-line material; many are free). As with IDL, one can write programs and applications also rather than just execute interactive commands. (Appendix A lists useful books, tutorials and on-line resources for learning the Python language itself as well as the specific modules addressed here.) Nevertheless, the focus will initially be almost entirely on interactive use. Later, as use becomes more sophisticated, more attention will be given to elements that are useful in writing scripts or applications (which themselves may be used as interactive commands).

For those with IDL experience: Appendix B compares Python and IDL to aid those trying to decide whether they should use Python; and Appendix C provides a mapping between IDL and Python array capabilities to help IDL users find corresponding ways to carry out the same actions. Appendix D (not available yet) will compare IDL plotting with matplotlib.

2 Prerequisites

Previous experience with Python of course is helpful, but is not assumed. Likewise, previous experience in using an array manipulation language such as IDL or matlab is helpful, but not required. Some familiarity with programming of some sort is necessary.

3 Practicalities

This tutorial series assumes that Python (v2.3 or later), numarray (v1.2.3 or later), matplotlib (v0.73.1 or later), pyfits (v0.98 or later), numdisplay (v1.0 or later), and ipython (v0.6.12 or later) are installed (Google to find the downloads if not already installed). All these modules will run on all standard Unix, Linux, Mac OS X and MS Windows platforms (PyRAF is not supported on MS Windows since IRAF does not run on that platform).

At STScI these are all available on Science cluster machines and the following describes how to set up your environment to access Python and these modules.

For the moment the PyFITS functions are available only on IRAFDEV. To make this version available either place IRAFDEV in your .envrc file or type `irafdev` at the shell level. Eventually this will be available for IRAFX as well.

If you have a Mac, and do not have all these items installed, follow the instructions on this web page: http://pyraf.stsci.edu/pyssg/macosx.html. At the very least you will need to do an update to get the latest PyFITS.

4 Tutorial 1: Reading and manipulating image data

4.1 Example session to read and display an image from a FITS file

The following illustrates how one can get data from simple fits file and display the data to DS9 or similar image display program. It presumes one has already started the image display program before starting Python. A short example of an interactive Python session is shown below (just the input commands, not what is printed out in return). The individual steps will be explained in detail in following sections.

```
>>> import pyfits                    # load FITS module
>>> from numarray import *           # load array module
>>> pyfits.info('pix.fits')          # show info about file
>>> im = pyfits.getdata('pix.fits') # read image data from file
>>> import numdisplay                # load image display module
>>> numdisplay.display(im)           # display to DS9
>>> numdisplay.display(im,z1=0,z2=300)  # repeat with new cuts
>>> fim = 1.*im                      # make float array
>>> bigvals = where(fim > 10)        # find pixels above threshld
                                    # log scale above threshold
>>> fim[bigvals] = 10*log(fim[bigvals]-10) + 10
>>> numdisplay.display(fim)
>>> hdr = pyfits.getheader('pix.fits')
>>> print hdr
>>> date = hdr['date']               # print header keyword value
>>> hdr['date'] = '4th of July'      # modify value
>>> hdr.update('flatfile','flat17.fits') # add new keyword flatfile
>>> pyfits.writeto('newfile.fits',fim,hdr) # create new fits file
>>> pyfits.append('existingfile.fits',fim, hdr)
>>> pyfits.update('existingfile.fits',fim, hdr, ext=3)
```

4.2 Starting the Python interpreter

The first step is to start the Python interpreter. There are several variants one can use. The plain Python interpreter is standard with every Python installation, but lacks many features that you will find in PyRAF or IPython. We recommend that you use one of those as an interactive environment (ultimately PyRAF will use IPython itself). For basic use they all pretty much do the same thing. IPython special features will be covered later. The examples will show all typed lines starting the line with the standard prompt of the Python interpreter (`>>>`) unless it is IPython (numbered prompt) or PyRAF (prompt: `-->`) being discussed. (Note that comments begin with `#`.) At the shell command line you type one of the following

```
python  # starts standard Python interpreter
ipython # starts ipython (enhanced interactive features)
pyraf   # starts PyRAF
```

4.3 Loading modules

After starting an interpreter, we need to load the necessary libraries. One can do this explicitly, as in this example, or do it within a start-up file. For now we'll do it explicitly. There is more than one way to load a library; each has its advantages. The first is the most common found in scripts:

```
>>> import pyfits
```

This loads the FITS I/O module. When modules or packages are loaded this way, all the items they contain (functions, variables, etc.) are in the "namespace" of the module and to use or access them, one must preface the item with the module name and a period, e.g., `pyfits.getdata()` to call the pyfits module getdata function.

For convenience, particularly in interactive sessions, it is possible to import the module's contents directly into the working namespace so prefacing the functions with the name of the module is not necessary. The following shows how to import the array module directly into the namespace:

```
>>> from numarray import *
```

There are other variants on importing that will be mentioned later.

4.4 Reading data from FITS files

One can see what a FITS file contains by typing:

```
>>> pyfits.info('pix.fits')
Filename: pix.fits
No.    Name         Type      Cards   Dimensions   Format
0    PRIMARY     PrimaryHDU      71  (512, 512)    Int16
```

The simplest way to access FITS files is to use the function getdata.

```
>>> im = pyfits.getdata('pix.fits')
```

If the fits file contains multiple extensions, this function will default to reading the data part of the primary Header Data Unit, if it has data, and if not, the data from the first extension. What is returned is, in this case, an image array (how tables are handled will be described in the next tutorial).

4.5 Displaying images

Much like IDL and Matlab, many things can be done with the array. For example, it is easy to find out information about the array: `im.shape` tells you about the dimensions of this array. The image can be displayed on a standard image display program such as ds9 (ximtool and SAOIMAGE will work too, so long as an 8-bit display is supported) using numdisplay:

```
>>> import numdisplay
>>> numdisplay.display(im)
```

As one would expect, one can adjust the image cuts:

```
>>> numdisplay.display(im,z1=0,z2=300)
```

Note that Python functions accept both positional style arguments or keyword arguments.

There are other ways to display images that will be covered in subsequent tutorials.

4.6 Array expressions

The next operations show that applying simple operations to the whole or part of arrays is possible.

```
>>> fim = 1.*im
```

creates a floating point version of the image.

```
>>> bigvals = where(fim > 10)
```

returns arrays indicating where in the fim array the values are larger than 10 (exactly what is being returned is glossed over for the moment). This information can be used to index the corresponding values in the array to use only those values for manipulation or modification as the following expression does:

```
>>> fim[bigvals] = 10*log(fim[bigvals]-10) + 10
```

This replaces all of the values that are larger than 10 in the array with a scaled log value added to 10

```
>>> numdisplay.display(fim)
```

The details on how to manipulate arrays will be the primary focus of this tutorial.

4.7 FITS headers

Looking at information in the FITS header is easy. To get the header one can use pyfits.getheader:

```
>>> hdr = pyfits.getheader('pix.fits')
```

(both header and data can be obtained using getdataheader; examples of such use will be covered in a later tutorial). To print out the whole header:

```
>>> print hdr
SIMPLE  =                    T / Fits standard
BITPIX  =                   16 / Bits per pixel
NAXIS   =                    2 / Number of axes
NAXIS1  =                  512 / Axis length
NAXIS2  =                  512 / Axis length
EXTEND  =                    F / File may contain extensions
```

[...]

```
CCDPROC = 'Apr 22 14:11 CCD processing done'
AIRMASS =    1.08015632629395   / AIRMASS
HISTORY 'KPNO-IRAF'
HISTORY '24-04-87'
HISTORY 'KPNO-IRAF'           /
HISTORY '08-04-92'            /
```

To get the value of a particular keyword:

```
>>> date = hdr['date']
>>> date
'2004-06-05T15:33:51'
```

To change an existing keyword:

```
>>> hdr['date'] = '4th of July'
```

To change an existing keyword or add it if it doesn't exist:

```
>>> hdr.update('flatfile','flat17.fits')
```

Where `flatfile` is the keyword name and `flat17.fits` is its value.

4.8 Writing data to FITS files

```
>>> pyfits.writeto('newfile.fits',fim)    # Creates barebone header
>>> pyfits.writeto('newfile.fits',fim,hdr) # User supplied header
```

or

```
>>> pyfits.append('existingfile.fits',fim, hdr)
```

or

```
>>> pyfits.update('existingfile.fits',fim, hdr, ext=3)
```

There are alternative ways of accessing FITS files that will be explained in a later tutorial that allow more control over how files are written and updated.

4.9 Some Python basics

It's time to gain some understanding of what is going on when using Python tools to do data analysis this way. Those familiar with IDL will see much similarity in the approach used. It may seem a bit more alien to those used to running programs or tasks in IRAF or similar systems.

4.9.1 Memory vs. data files

First it is important to understand that the results of what one does usually reside in memory rather than in a data file. With IRAF, most tasks that process data produce a new or updated data file (if the result is a small number of values it may appear in the printout or as a task parameter). In IDL or Python, one usually must explicitly write the results from memory to a data file. So results are volatile in the sense that they will be lost if the session is ended. The advantage of doing things this way is that applying a sequence of many operations to the data does not require vast amount of I/O (and the consequent cluttering of directories). The disadvantage is that very large data sets, where the size approaches the memory available, tend to need special handling (these situations will be addressed in a subsequent tutorial).

4.9.2 Python variables

Python is what is classified as a dynamically typed language (as is IDL). It is not necessary to specify what kind of value a variable is permitted to hold in advance. It holds whatever you want it to hold. To illustrate:

```
>>> value = 3
>>> print value
3
>>> value = ''hello''
```

Here we see the integer value of 3 assigned to the variable "value". Then the string "hello" is assigned to the same variable. To Python that is just fine. You are permitted to create new variables on the fly and assign whatever you please to them (and change them later). Python provides many tools (other than just "print") to find out what kind of value the variable contains. Variables

can contain simple types such as integers, floats, and strings, or much more complex objects such as arrays. Variable names contain letters, digits and '_'s and cannot begin with a digit. Variable names are case sensitive: name, Name, and NAME are all different variables.

Another important aspect to understand about Python variables is that they don't actually contain values, they refer to them (i.e., point), unlike IDL. So where one does:

```
>>> x = im # the image we read in
```

creates a new variable x but not a copy of the image. If one were to change a part of the image:

```
>>> im[5,5] = -999
```

The very same change would appear in the image referred to by x. This point can be confusing to some and everyone not used to this style will eventually stub their toes on it a few times. Generally speaking, when you want to copy data one must explicitly ask for a copy by some means. For example:

```
>>> x = im.copy()
```

(The odd style of this–for those not used to object oriented languages–will be explained next)

4.9.3 How does object oriented programming affect you?

While Python does support object-oriented programming very well, it does not require it to be used to write scripts or programs at all (indeed, there are many kinds of problems best not approached that way). It is quite simple to write Python scripts and programs without having to use object-oriented techniques (unlike Java and some other languages). In other words, just because Python supports object-oriented programming doesn't mean you have to use it that way or even that you should. That's good because the mere mention of object-oriented programming will give many astronomers the heebie-jeebies. That being said, there is a certain aspect of object oriented programming all users need to know about. While you are not required to write object-oriented programs, you will be required to use objects. Many Python libraries were written with the use of certain core objects in mind. Once you learn some simple rules, using these objects is quite simple. It's writing code that defines new kinds of objects that is what can be difficult to understand for newbies; not so for using them.

If one is familiar with structures (e.g., from C or IDL) one can think of objects as structures with bundled functions. Instead of functions, they are called methods. These methods in effect define what things are proper to do with the structure. For those not familiar with structures, they are essentially sets of variables that belong to one entity. Typically these variables can contain references to yet other structures (or for objects, other objects). An example illustrates much better than abstract descriptions.

```
>>> f = open('myfile.txt')
```

This opens a file and returns a file object. The object has attributes, things that describe it, and it has methods, things you can do with it. File objects have few attributes but several methods. Examples of attributes are:

```
>>> f.name
>>> f.mode
```

These are the name of the file associated with the file object and the read/write mode it was opened in. Note that attributes are identified by appending the name of the attribute to the object with a period. So one always uses .name for the file object's name. Here it is appended to the specific file object one wants the name for, that is `f`. If I had a different file object `bigfile`, then the name of that would be represented by `bigfile.name`.

Methods essentially work the same way except they are used with typical function syntax. One of a file object's methods is `readline`, which reads the next line of the file and returns it as a string. That is:

```
>>> f.readline()
'a line from the text file'
```

In this case, no arguments are needed for the method. The seek method is called with an argument to move the file pointer to a new place in the file. It doesn't return anything but instead performs an action to change the state of file object:

```
f.seek(1024) # move 1024 bytes from the file beginning.
```

Note the difference in style from the usual functional programming. A corresponding kind of call for seeking a file would be `seek(f,1024)`. Methods are implicitly supposed to work for the object they belong to. The method style of functions also means that it is possible to use the same method names for very different objects (particularly nice if they do the same basic action, like "close"). While the use of methods looks odd to those that haven't seen them before, they are pretty easy to get used to.

4.9.4 Errors and dealing with them

People make mistakes and so will you. Generally when mistakes are made with Python that the program did not expect to handle, an "Exception" is "raised". This essentially means the program has crashed and returned to the intepreter (it is not considered normal for a Python program to segfault or otherwise crash the Python interpeter; it can happen with bugs in C extensions–especially ones you may write–but it is very unusual for it to happen in standard Python libraries or Python code). When this happens you will see what is called a "traceback" which usually shows where in all the levels of the program, it

failed. While this can be lengthy and alarming looking, there is no need to get frightened. The most immediate cause of the failure will be displayed at the bottom (depending on the code and it's checking of user input, the original cause may or may not be as apparent). Unless you are interested in the programming details, it's usually safe to ignore the rest. To see your first traceback, let's intentionally make an error:

```
>>> f = pyfits.open(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "/usr/stsci/pyssg/py/pyfits.py", line 3483, in open
    ffo = _File(name, mode=mode, memmap=memmap)
  File "/usr/stsci/pyssg/py/pyfits.py", line 2962, in __init__
    self.__file = __builtin__.open(name, python_mode[mode])
TypeError: coercing to Unicode: need string or buffer, int found
```

The message indicates that a string (or suitable alternative) was expected and that an integer was found instead. The open function expects a filename, hence the exception.

The great majority of exceptions you will see will be due to usage errors. Nevertheless, some may be due to errors in the libraries or applications though, and should be reported if encountered (after ruling out usage errors).

4.10 Array basics

Arrays come with extremely rich functionality. A tutorial can only scratch the surface of the capabilities available. More details will be provided in later tutorials; the details can be found in the numarray manual.

4.10.1 Creating arrays

There are a few different ways to create arrays besides modules that obtain arrays from data files such as PyFITS.

```
>>> x = zeros((20,30))
```

creates a 20x30 array of zeros (default integer type; details on how to specify other types will follow). Note that the dimensions ("shape" in numarray parlance) are specified by giving the dimensions as a comma separated list within parentheses. The parentheses aren't necessary for a single dimension. As an aside, the parentheses used this way are being used to specify a Python tuple; more will be said about those in a later tutorial. For now you only need to imitate this usage.

Likewise one can create an array of 1's using the ones() function.

The arange() function can be used to create arrays with sequential values. E.g.,

```
>>> arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Note that that the array defaults to starting with a 0 value and does not include the value specified (though the array does have a length that corresponds to the argument)

Other variants:

```
>>> arange(10.)
array([ 0., 1., 2., 3., 4., 5., 6., 7., 8., 9])
>>> arange(3,10)
array([3, 4, 5, 6, 7, 8, 9])
>>> arange(1., 10., 1.1) # note trickiness
array([1. , 2.1, 3.2, 4.3, 5.4, 6.5, 7.6, 8.7, 9.8])
```

Finally, one can create arrays from literal arguments:

```
>>> print array([3,1,7])
[3 1 7]
>>> print array([[2,3],[4,4]])
[[2 3]
 [4 4]]
```

The brackets, like the parentheses in the zeros example above have a special meaning in Python which will be covered later (Python lists). For now, just mimic the syntax used here.

4.10.2 Array numeric types

numarray supports all standard numeric types. The default integer matches what Python uses for integers, usually 32 bit integers or what numarray calls Int32. The same is true for floats, i.e., generally 64-bit doubles called Float64 in numarray. The default complex type is Complex64. Many of the functions accept a type argument. For example

```
>>> zeros(3, Int8) # Signed byte
>>> zeros(3, type=UInt8) # Unsigned byte
>>> array([2,3], type=Float32)
>>> arange(4, type=Complex64)
```

The possible types are Int8, UInt8, Int16, UInt16, Int32, UInt32, Int64, UInt64, Float32, Float64, Complex32, Complex64. To find out the type of an array use the .type() method. E.g.,

```
>>> arr.type()
Float32
```

To convert an array to a different type use the astype() method, e.g,

```
>>> a = arr.astype(Float64)
```

4.10.3 Printing arrays

Interactively, there are two common ways to see the value of an array. Like many Python objects, just typing the name of the variable itself will print its contents (this only works in interactive mode). You can also explicitly print it. The following illustrates both approaches:

```
>>> x = arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8 9])
>>> print x
[0 1 2 3 4 5 6 7 8 9]
```

By default the array module limits the amount of an array that is printed out (to spare you the effects of printing out millions of values). For example:

```
>>> x = arange(1000000)
print x
[    0     1     2 ..., 999997 999998 999999]
```

If you really want to print out lots of array values, you can disable this feature or change the size of the threshold.

```
>>> import numarray.arrayprint as ap
>>> ap.summary_off() # disables limits on printing arrays
>>> ap.set_summary(threshhold=300, edge_items=3) # default
                          # threshhold=1000, edge_items=3
                          # (number at start and end to print)
```

4.10.4 Indexing 1-D arrays

As with IDL, there are many options for indexing arrays.

```
>>> x = arange(10)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Simple indexing:

```
>>> x[2] # 3rd element
2
```

Indexing is 0 based. The first value in the array is x[0]

Indexing from end:

```
>>> x[-2] # -1 represents the last element, -2 next to last...
8
```

Slices

To select a subset of an array:

```
>>> x[2:5]
array([2, 3, 4])
```

Note that the upper limit of the slice is not included as part of the subset! This is viewed as unexpected by newcomers and a defect. Most find this behavior very useful after getting used to it (the reasons won't be given here). Also important to understand is that slices are views into the original array in the same sense that references view the same array. The following demonstrates:

```
>>> y = x[2:5]
>>> y[0] = 99
>>> y
array([99, 3, 4])
>>> x
array([0, 1, 99, 3, 4, 5, 6, 7, 8, 9])
```

Changes to a slice will show up in the original. If a copy is needed use `x[2:5].copy()`

Short hand notation

```
>>> x[:5] # presumes start from beginning
array([ 0, 1, 99, 3, 4])
>>> x[2:] # presumes goes until end
array([99, 3, 4, 5, 6, 7, 8, 9])
>>> x[:] # selects whole dimension
array([0, 1, 99, 3, 4, 5, 6, 7, 8, 9])
```

Strides:

```
>>> x[2:8:3] # ''Stride'' every third element
array([99, 5])
```

Index arrays:

```
>>> x[[4,2,4,1]]
array([4, 99, 4, 1])
```

Using results of where function:

```
>>> x[where(x>5)]
array([99, 6, 7, 8, 9])
```

Mask arrays:

```
>>> m = x > 5
>>> m
array([0,0,1,0,0,0,1,1,1,1], type=Bool)
>>> x[m]
array([99, 6, 7, 8, 9])
```

4.10.5 Indexing multidimensional arrays

Before describing this in detail it is very important to note an item regarding multidimensional indexing that will certainly cause you grief until you become accustomed to it. ARRAY INDICES USE THE OPPOSITE CONVENTION AS FORTRAN, IDL AND IRAF REGARDING ORDER OF INDICES FOR MULTIDIMENSIONAL ARRAYS! There are long standing reasons for this and there are good reasons why this cannot be changed. Although it is possible to define arrays so that the traditional ordering applies, you are strongly encouraged to avoid this; it will only cause you problems later and you will eventually go insane as a result. Well, perhaps I exaggerate. But don't do it. Yes, we realize this is viewed by most as a tremendous defect. (If that prevents you from using or considering Python, so be it, but do weigh this against all the other factors before dismissing Python as a numerical tool out of hand).

```
>>> im = arange(24)
>>> im.shape=(4,6)
>>> im
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16, 17],
       [18, 19, 20, 21, 22, 23]])
```

To emphasize the point made in the previous paragraph, the index that represents the most rapidly varying dimension in memory is the 2nd index, not the first. We are used to that being the first dimension. Thus for most images read from a FITS file, what we have typically treated as the "x" index will be the second index. For this particular example, the location that has the value 8 in the array is im[1, 2].

```
>>> im[1, 2]
8
```

Partial indexing:

```
>>> im[1]
array([6, 7, 8, 9, 10, 11])
```

If only some of the indices for a multidimensional array are specified, then the result is an array with the shape of the "leftover" dimensions, in this case, 1-dimensional. The 2nd row is selected, and since there is no index for the column, the whole row is selected.

All of the indexing tools available for 1-D arrays apply to n-dimensional arrays as well (though combining index arrays with slices is not currently permitted). To understand all the indexing options in their full detail, read sections 4.6, 4.7 and 6 of the numarray manual.

4.10.6 Compatibility of dimensions

In operations involving combining (e.g., adding) arrays or assigning them there are rules regarding the compatibility of the dimensions involved. For example the following is permitted:

```
>>> x[:5] = 0
```

since a single value is considered "broadcastable" over a 5 element array. But this is not permitted:

```
>>> x[:5] = array([0,1,2,3])
```

since a 4 element array does not match a 5 element array.

The following explanation can probably be skipped by most on the first reading; it is only important to know that rules for combining arrays of different shapes are quite general. It is hard to precisely specify the rules without getting a bit confusing, but it doesn't take long to get a good intuitive feeling for what is and isn't permitted. Here's an attempt anyway: The shapes of the two involved arrays when aligned on their trailing part must be equal in value or one must have the value one for that dimension. The following pairs of shapes are compatible:

```
(5,4):(4,) -> (5,4)
(5,1):(4,) -> (5,4)
(15,3,5):(15,1,5) -> (15,3,5)
(15,3,5):(3,5) -> (15,3,5)
(15,1,5):(3,1) -> (15,3,5)
```

so that one can add arrays of these shapes or assign one to the other (in which case the one being assigned must be the smaller shape of the two). For the dimensions that have a 1 value that are matched against a larger number, the values in that dimension are simply repeated. For dimensions that are missing, the sub-array is simply repeated for those. The following shapes are not compatible:

```
(3,4):(4,3)
(1,3):(4,)
```

Examples:

```
>>> x = zeros((5,4))
>>> x[:,:] = [2,3,2,3]
>>> x
array([[2, 3, 2, 3],
       [2, 3, 2, 3],
       [2, 3, 2, 3],
       [2, 3, 2, 3],
```

```
       [2, 3, 2, 3]])
>>> a = arange(3)
>>> b = a[:] # different array, same data (huh?)
>>> b.shape = (3,1)
>>> b
array([[0],
       [1],
       [2]])
>>> a*b # outer product
array([[0, 0, 0],
       [0, 1, 2],
       [0, 2, 4]])
```

4.10.7 ufuncs

A ufunc (short for Universal Function) applies the same operation or function to all the elements of an array independently. When two arrays are added together, the add ufunc is used to perform the array addition. There are ufuncs for all the common operations and mathematical functions. More specialized ufuncs can be obtained from add-on libraries. All the operators have corresponding ufuncs that can be used by name (e.g., `add` for `+`). These are all listed in table below. Ufuncs also have a few very handy methods for binary operators and functions whose use are demonstrated here.

```
>>> x = arange(9)
>>> x.shape = (3,3)
>>> x
array([0, 1, 2],
      [3, 4, 5],
      [6, 7, 8]])
>>> add.reduce(x) # sums along the first index
array([9, 12, 15])
>>> add.reduce(x, axis=1) # sums along the 2nd index
array([3, 12, 21])
>>> add.accumulate(x) # cumulative sum along the first index
array([[0,  1,  2],
       [3,  5,  7],
       [9, 12, 15]])
>>> multiply.outer(arange(3),arange(3))
array([[0, 0, 0],
       [0, 1, 2],
       [0, 2, 4]])
```

Standard Ufuncs (with corresponding symbolic operators, when they exist, shown in parentheses)

add (+)	log	greater (>)
subtract (-)	log10	greater_equal (>=)
multiply (*)	cos	less (<)
divide (/)	arcos	less_equal (<=)
remainder (%)	sin	logical_and
absolute, abs	arcsin	logical_or
floor	tan	logical_xor
ceil	arctan	bitwise_and (&)
fmod	cosh	bitwise_or (|)
conjugate	sinh	bitwise_xor (^)
minimum	tanh	bitwise_not (~)
maximum	sqrt	rshift (>>)
power (**)	equal (==)	lshift (<<)
exp	not_equal (!=)	

Note that there are no corresponding Python operators for logical_ and and logical_ or. The Python "and" and "or" operators are NOT equivalent!

4.10.8 Array functions

There are many array utility functions. The following lists the more useful ones with a one line description. See the numarray manual for details on how they are used.

allclose: true if all elements within specified amount (between two arrays)

alltrue: test for all elements nonzero

argmax, argmin: return array with min/max locations for selected axis

argsort: returns indices of results of sort on an array

choose: fills specified array by selecting corresponding values from a set of arrays using integer selection array

clip: clip values of array

dot: dot product

compress: selects elements from array based on boolean array

concatenate: concatenate arrays

cumproduct: net cumulative product along specified axis

cumsum: accumulate array along specified axis

diagonal: returns diagonal of matrix

identity: returns identity matrix

indices: generate array with values corresponding to position of selected index of the array

innerproduct: guess

matrixmultiply: guess

outerproduct: guess

product: net product of elements along specified axis

ravel: creates a 1-d version of an array

repeat: fills given array with repeated copies of input array

resize: replicate or truncate array to new size

searchsorted: return indices of mapping values of an array into a monotonic array

sometrue: true if any element is true

sort: sort array elements along selected axis

sum: sum array along specified axis

swapaxes: switch indices for axis of array (doesn't actually move data, just maps indices differently)

trace: compute trace of matrix

transpose: transpose indices of array (doesn't actually move data, just maps indices differently)

where: find "true" locations in array

4.10.9 Array methods

Arrays have several methods. They are used as methods are with any object. For example (using the array from the previous example):

```
>>> # sum all array elements
>>> x.sum() # the L indicates a Python Long integer
36L
```

The following lists all the array methods that exist (a number are equivalent to array functions; these have no summary description shown):

argmax

argmin

argsort

astype: copy array to specified numeric type

byteswap: perform byteswap on data in place

byteswapped: return byteswapped copy of array

conjugate: complex conjugate

copy: produce copied version of array (instead of view)

diagonal

info: print info about array

isaligned: are data elements guaranteed aligned with memory?

isbyteswapped: are data elements in native processor order?

iscontiguous: are data elements contiguous in memory?

is_c_array: are data elements aligned, not byteswapped, and contiguous?

is_fortran_contiguous: are indicies defined to follow Fortran conventions?

is_f_array: are indices defined to follow Fortran conventions and data are aligned and not byteswapped

itemsize: size of data element in bytes

max: maximum value in array

min: minimum value in array

nelements: total number of elements in array

new: returns new array of same type and size (data uninitialized)

repeat

resize

size: same as nelements

type: returns type of array

typecode: returns corresponding typecode character used by Numeric

tofile: write binary data to file

tolist: convert data to Python list format

tostring: copy binary data to Python string

transpose

stddev: standard deviation

sum: sum of all elements

swapaxes

togglebyteorder: change byteorder flag without changing actual data byte-order

trace

view: returns new array object using view of same data

4.10.10 Array attributes:

shape: returns shape of array

flat: returns view of array treating it as 1-dimensional. Doesn't work if array is not contiguous

real: return real component of array (exists for all types)

imag, imaginary: return imaginary component (exists only for complex types)

4.11 Example

The following example shows how to use some of these tools. The idea is to select only data within a given radius of the center of the galaxy displayed and compute the total flux within that radius using the built-in array facilities.

First we will create a mask for all the pixels within 50 pixels of the center. We begin by creating x and y arrays that whose respective values indicate the x and y locations of pixels in the arrays:

```
# first find location of maximum in image
y, x = indices(im.shape, type=Float32) # note order of x, y!
# after finding peak at 257,258 using ds9
x = x-257 # make peak have 0 value
y = y-258
radius = sqrt(x**2+y**2)
mask = radius < 50
display mask*im
(mask*im).sum() # sum total of masked image
# or
im[where(mask)].sum() # sum those points within the radius
# look Ma, no loops!
```

4.12 Exercises

The needed data for these exercises can be downloaded from stsdas.stsci.edu/python.

1. Start up Python (Python, PyRAF, or IPython). Type `print 'hello world'`. Don't go further until you master this completely.

2. Read pix.fits, find the value of the OBJECT keyword, and print all the image values in the 10th column of the image.

3. Scale all the values in the image by 2.3 and store in a new variable. Determine the mean value of the scaled image

4. Save the center 128x128 section of the scaled image to a new FITS file using the same header. It won't be necessary to update the NAXIS keywords of the header; that will be handled automatically by PyFITS.

5. Extra credit: Create an image (500x500) where the value of the pixels is a Gaussian-weighted sinusoid expressed by the following expression:

   $\sin(x/\pi)e^{-((x-250)^2+(y-250)^2)/2500.}$

   where x represents the x pixel location and likewise for y. Display it. Looking at the numarray manual, find out how to perform a 2-D FFT and display the absolute value of the result.

Acknowledgements

Thanks to Phil Hodge, JC Hsu, Todd Miller, and Dave Grumm for valuable comments on previous versions of this document.

Appendix A: Python Books, Tutorials, and On-line Resources

The Python web site (www.python.org) contains pointers to a great deal of information about Python including its standard documentation, conferences, books, tutorials, etc. In particular, the following links will contain up-to-date lists of books and tutorials:

- Books: www.python.org/moin/PythonBooks
- Tutorials: www.python.org/doc/Intros.html (all online)

Books of note for beginners that have some programming experience:

- *Learning Python* (2nd ed) Lutz & Ascher: Probably the standard introductory book
- *The Quick Python Book* by Harms & McDonald: some prefer the style of this one
- *Python: Visual QuickStart Guide* by Fehily: and others this one
- *Dive Into Python: Python for Experienced Programmers* by Pilgrim (free on-line version also available)

On-line tutorials or books:

- *An Introduction to Python* by van Rossum: The original tutorial
- *A Byte of Python by Swaroop*: essentially an on-line book.
- *A Quick Tour of Python* by Greenfield and White: http://stsdas.stsci.edu/pyraf/doc/python_quick_tour (A bit dated but comparatively brief). PDF also available.

Reference books:

- *Python Essential Reference* (2nd ed.) by Beazley
- *Python in a Nutshell* by Martelli
- Python Library Documentation (available free on-line)

Mailing lists:

- astropy: For astronomical packages (e.g. PyRAF, PyFITS, numdisplay): http://www.scipy.net/mailman/listinfo/astropy
- numarray: http://sourceforge.net/mail/?group_id=1369
- matplotlib: http://sourceforge.net/mail/?group_id=80706

- ipython: http://www.scipy.net/pipermail/ipython-user/

Manuals for numarray, PyFITS, PyRAF, ipython, and matplotlib are all available. Google PyFITS, numarray or PYRAF with "manual" to find those. The matplotlib User Guide is available on a link from the home page (Google "matplotlib"). The ipython manual is available from ipython.scipy.org. The numdisplay instructions are available at: http://stsdas.stsci.edu/numdisplay/doc/numdisplay_help.html

Appendix B: Why would I switch from IDL to Python (or not)?

We do not claim that all, or even most, current IDL users should switch to using Python now. IDL suits many people's needs very well and we recognize that there must be a strong motivation for starting to use Python over IDL. This appendix will present the pros and cons of each so that users can make a better informed decision about whether they should consider using Python. At the end we give a few cases where we feel users should give serious consideration to using Python over IDL.

Pros and Cons of each

These are addressed in a comparative sense. Attributes that both share, e.g.., that they are interpreted and relatively slow for very simple operations, are not listed.

Pros of IDL:

- Mature many numerical and astronomical libraries available
- Wide astronomical user base
- Numerical aspect well integrated with language itself
- Many local users with deep experience
- Faster for small arrays
- Easier installation
- Good, unified documentation
- Standard GUI run/debug tool
- Single widget system (no angst about which to choose or learn)
- SAVE/RESTORE capability
- Use of keyword arguments as flags more convenient

Cons of IDL:

- Narrow applicability, not well suited to general programming
- Slower for large arrays
- Array functionality less powerful
- Table support poor

- Limited ability to extend using C or Fortran, such extensions hard to distribute and support
- Expensive, sometimes problem collaborating with others that don't have or can't afford licences.
- Closed source (only RSI can fix bugs)
- Very awkward to integrate with IRAF tasks
- Memory management more awkward
- Single widget system (useless if working within another framework)
- Plotting:
  - Awkard support for symbols and math text
  - Many font systems, portability issues (v5.1 alleviates somewhat)
  - not as flexible or as extensible
  - plot windows not intrinsically interactive (e.g., pan & zoom)

Pros of Python:

- Very general and powerful programming language, yet easy to learn. Strong, but optional, Object Oriented programming support
- Very large user and developer community, very extensive and broad library base
- Very extensible with C, C++, or Fortran, portable distribution mechanisms available
- Free; non-restrictive license; Open Source
- Becoming the standard scripting language for astronomy
- Easy to use with IRAF tasks
- Basis of STScI application efforts
- More general array capabilities
- Faster for large arrays, better support for memory mapping
- Many books and on-line documentation resources available (for the language and its libraries)
- Better support for table structures
- Plotting

  - framework (matplotlib) more extensible and general
  - Better font support and portability (only one way to do it too)
  - Usable within many windowing frameworks (GTK, Tk, WX, Qt...)
  - Standard plotting functionality independent of framework used
  - plots are embeddable within other GUIs
  - more powerful image handling (multiple simultaneous LUTS, optional resampling/rescaling, alpha blending, etc)
- Support for many widget systems
- Strong local influence over capabilities being developed for Python

Cons of Python:

- More items to install separately
- Not as well accepted in astronomical community (but support clearly growing)
- Scientific libraries not as mature:
  - Documentation not as complete, not as unified
  - Numeric/numarray split
  - Not as deep in astronomical libraries and utilities
  - Not all IDL numerical library functions have corresponding functionality in Python
- Some numeric constructs not quite as consistent with language (or slightly less convenient than IDL)
- Array indexing convention "backwards"
- Small array performance slower
- No standard GUI run/debug tool
- Support for many widget systems (angst regarding which to choose)
- Current lack of function equivalent to SAVE/RESTORE in IDL
- matplotlib does not yet have equivalents for all IDL 2-D plotting capability (e.g., surface plots)
- Use of keyword arguments used as flags less convenient
- Plotting:
  - comparatively immature, still much development going on
  - missing some plot type (e.g., surface)
  - 3-d capability requires VTK

Specific cases

Here are some specific instances where using Python provides strong advantages over IDL

- Your processing needs depend on running a few hard to replicate IRAF tasks, but you don't want to do most of your data manipulation in IRAF, but would rather write your own IDL-style programs to do so (and soon other systems will be accessible from Python, e.g., MIDAS, ALMA, slang, etc)
- You have algorithms that cannot be efficiently coded in IDL. They likely won't be efficiently coded in Python either, but you will find interfacing the needed C or Fortran code easier, more flexible, more portable, and distributable. (Question: how many distributed IDL libraries developed by 3rd parties include C or Fortran code?) Or you need to wrap existing C libraries (Python has many tools to make this easier to do).
- You do work on algorithms that may migrate into STSDAS packages. Using Python means that your work will be more easily adapted as a distributed and supported tool.
- You wish to integrate data processing with other significant non-numerical processing such as databases, web page generation, web services, text processing, process control, etc.
- You want to learn object-oriented programming and use it with your data analysis. (But you don't need to learn object-oriented programming to do data analysis in Python.)
- You want to be able to use the same language you use for data analysis for most of your other scripting and programming tasks.
- Your boss makes you.
- You want to be a cool, with-it person.
- You are honked off at ITT Space Systems/RSI.

Obviously using a new language and libraries entails time spent learning. Despite what people say, it's never that easy, especially if one has a lot of experience and code invested in an existing language. If you don't have any strong motivations to switch, you should probably wait.

Appendix C: IDL/numarray feature mapping

IDL	Python Equivalent
`.run`	`import` (.py assumed) `reload(module)` # to recompile/re-execute ipython: `run`
`@<filename>`	`execfile('fileame')` ipython: `run`
`exit`	control-D (MS windows: control-Z) ipython: `Quit` or `Exit`
up-arrow (command recall)	up-arrow
Operators	
`<,> (clipping)`	Currently no operator equivalent. The functional equivalent is `choose(a<100, (a, 100.))`
`a < 100.`	
`a > 100.`	
`MOD`	`%`
`# (matrix multiply)`	`multiply.outer()` is equivalent for some applications, `matrixmultiply()` for others.
`^`	`**`
Boolean operators	
`AND`	no operator equivalent Python and operator is not the same!
`OR`	no operator equivalent Python or operator is not the same!
`NOT`	no operator equivalent Python not operator is not the same! Note: Python bitwise operators much like IDL boolean operators
Comparison operators	
`EQ`	`==`
`NE`	`!=`
`LT`	`<`
`LE`	`<=`
`GT`	`>`
`GE`	`>=`
Bitwise operators	
`AND`	`&`
`OR`	`|`
`XOR`	`^`
`NOT`	`~`
`ishift(a,1)`	`a<<1`
`ishift(a,-1)`	`a>>1`

Slicing and indexing	**Note: order of indices is opposite!**
	`a[i,j]` (IDL) equivalent to `a[j,i]` (Python)
`i:j`	`i:j` (j element not part of slice)
`i:*`	`i:`
`*:i`	`:i`
`a[i,*]`	`a[:,i]`
`a[i:j:s]`	`a[i:j:s]`
	empty arrays and slices permitted (E.g., `a[0:0]` is an array of length 0)
	`-i` (indexing from end of array)
	`...` (fills in unspecified dimensions)
	`NewAxis` (add new axis to array shape as part of subscript to match shapes)
	Slicing does not create copy of data, but a new view of data.
Index arrays	
`newarr = arr[indexarr]`	`newarr = arr[indexarr]`
`arr[indexarr] = valuearr`	`arr[indexarr] = valuearr`
Array Creation	
`fltarr()`	`array(seq, [type])` to create from existing sequence
`dblarr()`	`zeros(shape, [type])` to create 0 filled array
`complexarr()`	`ones(shape, [type])` to create 1 filled array
`intarr()`	chararray for fixed length strings
`longarr()`	
`bytarr()`	
`make_arr()`	
`strarr()`	
`findgen()`	`arange(size, [type]))`
`dindgen()`	`arange(start, end, [type])`
`indgen()`	`arange(start, end, step, [type])`
`lindgen()`	no equivalent for strings
`sindgen()`	
`replicate()`	`repeat` (more general)
Array Conversions	
`byte(arrayvar)`	`arrayvar.astype(<type>)`
`fix(arrayvar)`	
`long(arrayvar)`	
`float(arrayvar)`	
`double(arrayvar)`	
`complex(arrayvar)`	

Array Manipulation

`reform()`	`reshape()` [also `arrayvar.flat()` and `ravel()`, or changing the shape attribute]
`sort()`	`argsort()` `[i.e., indices needed to sort]`
`arr(sort(arr))`	`sort(arr)` `[i.e., sorted array]`
`max(arrayvar)`	`arrayvar.max()`
`min(arrayvar)`	`arrayvar.min()`
`where()`	`where()`
`arr(where(condition))`	`arr[where(condition)]`
`transpose()`	`transpose()`
`[a,b]` (array concatenation)	`concatenate()`

Array shape behavior

shape mismatches result in the smaller of the two	If shapes do not follow broadcast rules, error generated. No shape truncation performed ever.

Numeric types

`byte`	`Int8`
	`UInt8` (unsigned)
`int`	`Int16`
`long`	`Int32`
`float`	`Float32`
`double`	`Float64`
`complex`	`Complex64`
	`Complex128`
	`UInt16`
	`UInt32`
	`Int64`
	`UInt64`

Math Functions

`abs()`	`abs()`
`acos()`	`arccos()`
`alog()`	`log()`
`alog10()`	`log10()`
`asin()`	`arcsin()`
`atan()`	`arctan()`
`atan(y,x)`	`arctan2()`
`ceil()`	`ceil()`
`conj()`	`conjugate()`
`cos()`	`cos()`
`cosh()`	`cosh()`
`exp()`	`exp()`
`floor()`	`floor()`
`imaginary()`	`complexarr.imag` (`.real` for real component)

`invert()`	Matrix module
`ishift()`	`right_shift(), left_shift()`
`round()`	`round()`
`sin()`	`sin()`
`sinh()`	`sinh()`
`sqrt()`	`sqrt()`
`tan()`	`tan()`
`tanh()`	`tanh()`
`fft()`	`fft` (fft module)
`convol()`	`convolve()` (convolve module)
`randomu()`	`random(), uniform()` (random_array module)
`randomn()`	`normal()` (random_array module)

Programming

`execute()`	`exec()`
`n_elements(arrayvar)`	`arrayvar.nelements()`
`n_params()`	closest equivalent is `len(*args)`. Ability to supply default values for parameters replaces some uses of `n_params()`
`size()`	`.shape` attribute, `.type()` method
`wait`	`time.sleep` (time module)

Appendix D: IDL plotting compared with matplotlib

Forthcoming.