

Using Python for Interactive Data Analysis

Perry Greenfield
Robert Jedrzejewski
Vicki Laidler
Space Telescope Science Institute

9th May 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 compares 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 a 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 numpy 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 threshold
                                   # 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` (the following examples presume that the image display program has already been started):

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

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 at the same time using `getdata` with an optional header keyword argument (the ability to assign to two variables at once will be explained a bit more in a later tutorial; it's particularly useful when functions return more than one thing):

```
>>> data, hdr = pyfits.getdata('pix.fits', header=True)
```

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.

Special methods are available to add history, comment or blank cards (see Tutorial 3 for examples).
When

4.8 Writing data to FITS files

```
>>> 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 `'_'` 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
'myfile.txt'
>>> f.mode
'r'
```

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 interpreter (it is not considered normal for a Python program to segfault or otherwise crash the Python interpreter; 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).

Unlike IDL, exceptions do not leave you at the level of the program that caused them. Enabling that behavior is possible, and will be discussed in tutorial 4.

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 (which behaves somewhat differently than for IDL; covered in tutorial 3):

```
>>> 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 we 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 (/)	arcsin	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 to these respective ufuncs!

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 `numpy` manual for details on how they are used. Arguments shown with `argument=value` indicate what the default value is if called without a value for that argument.

`all(a)`: are all elements of array nonzero

`allclose(a1, a2, rtol=1.e-5, atol=1.e-8)`: true if all elements within specified amount (between two arrays)

`alltrue(a, axis=0)`: are all elements nonzero along specified axis true.

`any(a)`: are any elements of an array nonzero

`argmax(a, axis=-1)`, `argmin(a,axis=-1)`: return array with min/max locations for selected axis

`argsort(a, axis=-1)`: returns indices of results of sort on an array

`choose(selector, population, clipmode=CLIP)`: fills specified array by selecting corresponding values from a set of arrays using integer selection array (population is a tuple of arrays; see tutorial 2)

`clip(a, amin, amax)`: clip values of array *a* at values *amin*, *amax*

`dot(a1, a2)`: dot product of arrays *a1* & *a2*

`compress(condition, a, axis=0)`: selects elements from array *a* based on boolean array *condition*

`concatenate(arrays, axis=0)`: concatenate arrays contained in sequence of arrays *arrays*

`cumproduct(a, axis=0)`: net cumulative product along specified axis

`cumsum(a, axis=0)`: accumulate array along specified axis

`diagonal(a, offset=0, axis1=0, axis2=1)`: returns diagonal of 2-d matrix with optional offsets.

`fromfile(file, type, shape=None)`: Use binary data in file to form new array of specified type.

`fromstring(datastring, type, shape=None)`: Use binary data in *datastring* to form new array of specified shape and type

`identity(n, type=None)`: returns identity matrix of size *nxn*.

`indices(shape, type=None)`: generate array with values corresponding to position of selected index of the array

`innerproduct(a1, a2)`: guess

`matrixmultiply(a1, a2)`: guess

`outerproduct(a1, a2)`: guess

`product(a, axis=0)`: net product of elements along specified axis

`ravel(a)`: creates a 1-d version of an array

`repeat(a, repeats, axis=0)`: generates new array with repeated copies of input array *a*

`resize(a, shape)`: replicate or truncate array to new shape

`searchsorted(bin, a)`: return indices of mapping values of an array *a* into a monotonic array *bin*

`sometrue(a, axis=0)`: are any elements along specified axis true

`sort(a, axis=-1)`: sort array elements along selected axis

`sum(a, axis=0)`: sum array along specified axis

`swapaxes(a, axis1, axis2)`: switch indices for axis of array (doesn't actually move data, just maps indices differently)

`trace(a, offset=0, axis1=0, axis2=1)`: compute trace of matrix *a* with optional offset.

`transpose(a, axes=None)`: transpose indices of array (doesn't actually move data, just maps indices differently)

`where(a)`: find "true" locations in array *a*

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 for an array object *a* (a number are equivalent to array functions; these have no summary description shown):

- a.argmax(axis=-1)*
- a.argmin(axis=-1)*
- a.argsort(axis=-1)*
- a.astype(type)*: copy array to specified numeric type
- a.byteswap()*: perform byteswap on data in place
- a.byteswapped()*: return byteswapped copy of array
- a.conjugate()*: complex conjugate
- a.copy()*: produce copied version of array (instead of view)
- a.diagonal()*
- a.info()*: print info about array
- a.isaligned()*: are data elements guaranteed aligned with memory?
- a.isbyteswapped()*: are data elements in native processor order?
- a.iscontiguous()*: are data elements contiguous in memory?
- a.is_c_array()*: are data elements aligned, not byteswapped, and contiguous?
- a.is_fortran_contiguous()*: are indices defined to follow Fortran conventions?
- a.is_f_array()*: are indices defined to follow Fortran conventions and data are aligned and not byteswapped
- a.itemsize()*: size of data element in bytes
- a.max(type=None)*: maximum value in array
- a.min()*: minimum value in array
- a.nelements()*: total number of elements in array
- a.new()*: returns new array of same type and size (data uninitialized)
- a.repeat(a, repeats, axis=0)*:
- a.resize(shape)*:
- a.size()*: same as nelements
- a.type()*: returns type of array
- a.typecode()*: returns corresponding typecode character used by Numeric
- a.tofile(file)*: write binary data to file
- a.tolist()*: convert data to Python list format

`a.tostring()`: copy binary data to Python string
`a.transpose(axes=-1)`: transpose array
`a.stddev()`: standard deviation
`a.sum()`: sum of all elements
`a.swapaxes(axis1,axis2)`
`a.togglebyteorder()`: change byteorder flag without changing actual data byteorder
`a.trace()`
`a.view()`: returns new array object using view of same data

4.10.10 Array attributes:

`a.shape`: returns shape of array
`a.flat`: returns view of array treating it as 1-dimensional. Doesn't work if array is not contiguous
`a.real`: return real component of array (exists for all types)
`a.imag`, `a.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 stdas.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.

5 Tutorial 2: Reading and plotting spectral data

In this tutorial we will cover some simple plotting commands using `Matplotlib`, a Python plotting package developed by John Hunter of the University of Chicago. We will also talk about reading FITS tables, delve a little deeper into some of Python's data structures, and use a few more of Python's features that make coding in Python so straightforward.

5.1 Example session to read spectrum and plot it

The sequence of commands below represent reading a spectrum from a FITS table and using `matplotlib` to plot it. Each step will be explained in more detail in following subsections.

```
>>> import pyfits
>>> from pylab import *           # import plotting module
>>> pyfits.info('fuse.fits')
>>> tab = pyfits.getdata('fuse.fits') # read table
>>> tab.names                      # names of columns
>>> tab.formats                    # formats of columns
>>> flux = tab.field('flux')       # reference flux column
>>> wave = tab.field('wave')
>>> flux.shape                     # show shape of flux column array
>>> plot(wave, flux)               # plot flux vs wavelength
                                # add xlabel using symbols for lambda/angstrom
>>> xlabel(r'$\lambda$ (\angstrom)$', size=13)
>>> ylabel('Flux')
# Overplot smoothed spectrum as dashed line
>>> from numpy.convolve import boxcar
>>> sflux = boxcar(flux.flat, (100,)) # smooth flux array
>>> plot(wave, sflux, '--r', hold=True) # overplot red dashed line
>>> subwave = wave.flat[::100]       # sample every 100 wavelengths
>>> subflux = flux.flat[::100]
>>> plot(subwave, subflux, 'og')     # overplot points as green circles
>>> errorbar(subwave, subflux, yerr=0.05*subflux, fmt='.k')
>>> legend(('unsmoothed', 'smoothed', 'every 100'))
>>> text(1007, 0.5, 'Hi There')
                                # save to png and postscript files
>>> savefig('fuse.png')
>>> savefig('fuse.ps')
```

5.2 An aside on how Python finds modules

When you import a module, how does Python know where to look for it? When you start up Python, there is a search path defined that you can access using the `path` attribute of the `sys` module. So:

```
>>> import sys
>>> sys.path
['', 'C:\\WINNT\\system32\\python23.zip',
 'C:\\Documents and Settings\\rij\\SSB\\demo',
 'C:\\Python23\\DLLs', 'C:\\Python23\\lib',
 'C:\\Python23\\lib\\plat-win',
 'C:\\Python23\\lib\\lib-tk',
 'C:\\Python23',
 'C:\\Python23\\lib\\site-packages ',
 'C:\\Python23\\lib\\site-packages\\Numeric',
 'C:\\Python23\\lib\\site-package s\\gtk-2.0',
```

```
'C:\\Python23\\lib\\site-packages\\win32',
'C:\\Python23\\lib\\site_packages\\win32\\lib',
'C:\\Python23\\lib\\site-packages\\Pythonwin']
```

This is a list of the directories that Python will search when you import a module. If you want to find out where Python actually found your imported module, the `__file__` attribute shows the location:

```
>>> pyfits.__file__
'C:\\Python23\\lib\\site-packages\\pyfits.pyc'
```

Note the double `'\'` characters in the file specifications; Python uses `\` as its escape character (which means that the following character is interpreted in a special way. For example, `\n` means “newline”, `\t` means “tab” and `\a` means “ring the terminal bell”). So if you really *want* a backslash, you have to escape it with another backslash. Also note that the extension of the `pyfits` module is `.pyc` instead of `.py`; the `.pyc` file is the bytecode compiled version of the `.py` file that is automatically generated whenever a new version of the module is executed.

5.3 Reading FITS table data

As well as reading regular FITS images, PyFITS also reads tables as arrays of records (rearrays in numarray parlance). These record arrays may be indexed just like numeric arrays though numeric operations cannot be performed on the record arrays themselves. All the columns in a table may be accessed as arrays as well.

```
>>> import pyfits
```

Assuming a FITS table of FUSE data in the current directory with the imaginative name of `'fuse.fits'`

```
>>> pyfits.info('fuse.fits')
Filename: fuse.fits
No.    Name          Type          Cards  Dimensions  Format
0     PRIMARY      PrimaryHDU    365    ()          Int16
1     SPECTRUM     BinTableHDU   35     1R x 7C     [10001E,
10001E, 10001E, 10001J, 10001E, 10001E, 10001I]
>>> tab = pyfits.getdata('fuse.fits') # returns table as record array
```

PyFITS record arrays have a `names` attribute that contains the names of the different columns of the array (there is also a `format` attribute that describes the type and contents of each column).

```
>>> tab.names
['WAVE', 'FLUX', 'ERROR', 'COUNTS', 'WEIGHTS', 'BKGD', 'QUALITY']
>>> tab.formats
['10001Float32', '10001Float32', '10001Float32', '10001Float32',
'10001Float32', '10001Float32', '10001Int16']
```

The latter indicates that each column element contains a 10001 element array of the types indicated.

```
>>> tab.shape
(1,)
```

The table only has one row. Each of the columns may be accessed as its own array by using the field method. Note that the shape of these column arrays is the combination of the number of rows and the size of the columns. Since in this case the columns contain arrays, the result will be two-dimensional (albeit with one of the dimensions only having length one).

```
>>> wave = tab.field('wave')
>>> flux = tab.field('flux')
>>> flux.shape
(1, 10001)
```

The arrays obtained by the field method are not copies of the table data, but instead are views into the record array. If one modifies the contents of the array, then the table itself has changed. Likewise, if a record (i.e., row) of the record array is modified, the corresponding column array will change. This is best shown with a different table:

```
>>> tab2 = getdata('table2.fits')
>>> tab2.shape # how many rows?
(3,)
>>> tab2
array(
  [('M51', 13.5, 2),
   ('NGC4151', 5.7999999999999998, 5),
   ('Crab Nebula', 11.119999999999999, 3)],
  formats=['1a13', '1Float64', '1Int16'],
  shape=3,
  names=['targname', 'flux', 'nobs'])
>>> col3 = tab2.field('nobs')
>>> col3
array([2, 5, 3], type=Int16)
>>> col1[2] = 99
>>> tab2
array(
  [('M51', 13.5, 2),
   ('NGC4151', 5.7999999999999998, 5),
   ('Crab Nebula', 11.119999999999999, 99)],
  formats=['1a13', '1Float64', '1Int16'],
  shape=3,
  names=['targname', 'flux', 'nobs'])
```

Numeric column arrays may be treated just like any other numarray array. Columns that contain character fields are returned as character arrays (with their own methods, described in the PyFITS User Manual)

Updated or modified tables can be written to FITS files using the same functions as for image or array data.

5.4 Quick introduction to plotting

The package `matplotlib` is used to plot arrays and display image data. This section gives a few examples of how to make quick plots. More examples will appear later in the tutorial (these plots assume that the `.matplotlibrc` file has been properly configured; the default version at STScI has been set up that way. There will be more information about the `.matplotlibrc` file later in the tutorial).

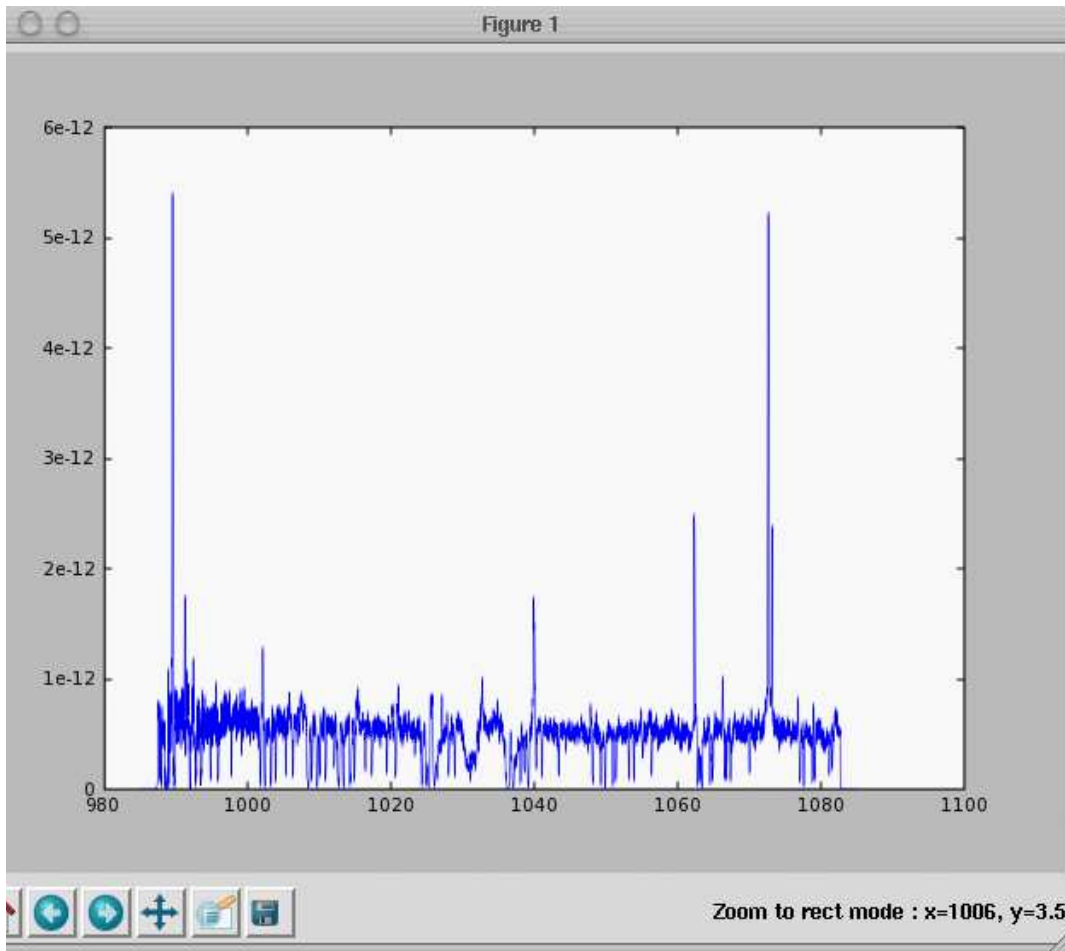
First, we must import the functional interface to `matplotlib`

```
>>> from pylab import *
```

5.4.1 Simple x-y plots

To plot flux vs wavelength:

```
>>> plot(wave, flux)
[<matplotlib.lines.Line2D instance at 0x02A07440>]
```



Note that the resulting plot is interactive. The toolbar at the bottom is used for a number of actions. The button with arrows arranged in a cross pattern is used for panning or zooming the plot. In this mode the zooming is accomplished by using the middle mouse button; dragging it in the x direction affects the zoom in that direction and likewise for the y direction. The button with a magnifying glass and rectangle is used for the familiar zoom to rectangle (use the left mouse button to drag define a rectangle that will be the new view region for the plot). The left and right arrow buttons can be used to restore different views of the plot (a history is kept of every zoom and pan). The button with a house will return it the the original view. The button with a diskette allows one to save the plot to a .png or postscript file. You can resize the window and the plot will re-adjust to the new window size.

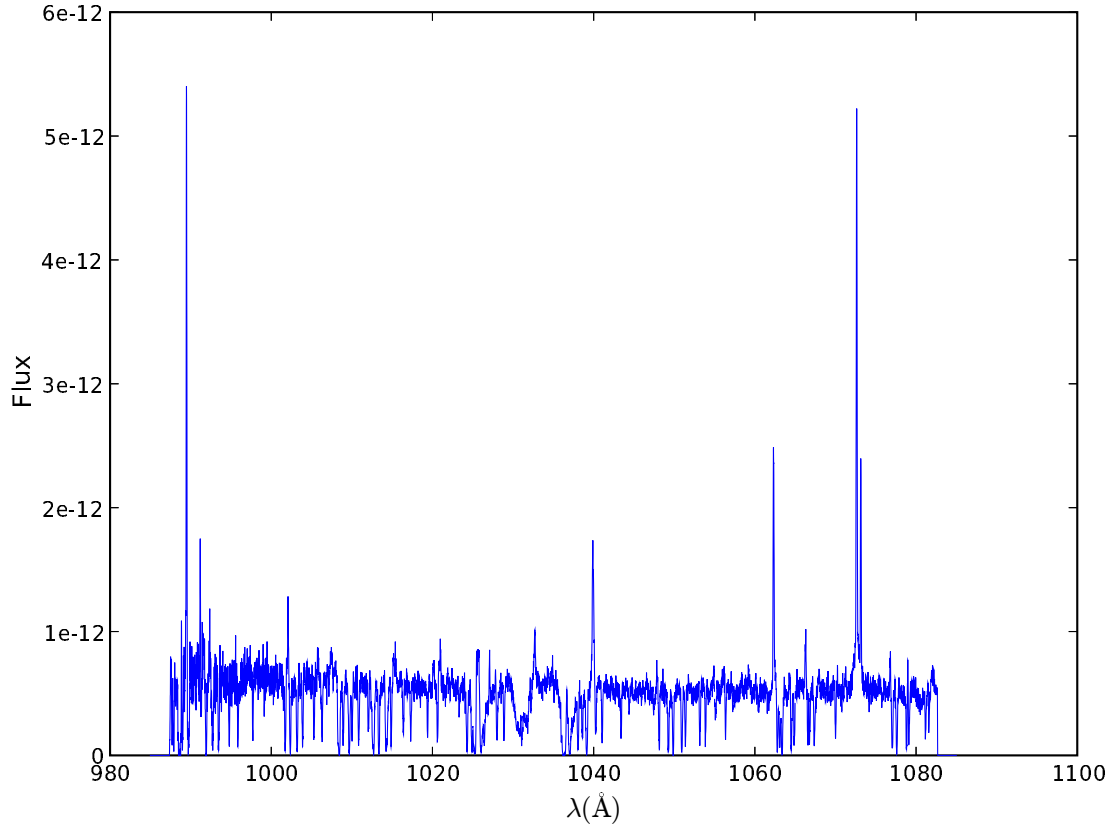
Also note that this and many of the other pylab commands result in a cryptic printout. That's because these function calls return a value. In Python when you are in an interactive mode, the act of entering a value at the command line, whether it is a literal value, evaluated expression, or return value of a function, Python attempts to print out some information on it. Sometimes that shows you the value of the object (if it is simple enough) like for numeric values or strings, or sometimes it just shows the type of the object, which is what is being shown here. The functions return a value so that you can assign it to a variable to manipulate the plot later (it's not necessary to do that though, though for now it does help keep the screen uncluttered). We are likely to change the behavior of the object so that nothing is printed (even though it is still returned) so your session screen will not be cluttered with these messages.

5.4.2 Labeling plot axes

It is possible to customize plots in many ways. This section will just illustrate a few possibilities

```
>>> xlabel(r'$\lambda$ (\angstrom)$', size=13)
<matplotlib.text.Text instance at 0x029A9F30>
```

```
>>> ylabel('Flux')
<matplotlib.text.Text instance at 0x02A07BC0>
```



One can add standard axis labels. This example shows that it is possible to use special symbols using \TeX notation.

5.4.3 Overplotting

Overplots are possible. First we make a smoothed spectrum to overplot.

```
>>> from numpy.convolve import boxcar # yet another way to import functions
>>> sflux = boxcar(flux.flat, (100,)) # smooth flux array using size 100 box
>>> plot(wave, sflux, '--r', hold=True) # overplot red dashed line
[<matplotlib.lines.Line2D instance at 0x0461FC60>]
```

This example shows that one can use the `hold` keyword to overplot, and how to use different colors and linestyles. This case uses a terse representation (`--` for dashed and `r` for red) to set the values, but there are more verbose ways to set these attributes. As an aside, the functions that matplotlib uses are closely patterned after Matlab. Next we subsample the array to overplot circles and error bars for just those points.

```
>>> subwave = wave.flat[::100] # sample every 100 wavelengths
>>> subflux = flux.flat[::100]
>>> plot(subwave, subflux, 'og', hold=True) # overplot points as green circles
[<matplotlib.lines.Line2D instance at 0x046EBE40>]
>>> error = tab.field('error')
>>> suberror = error.flat[::100]
```

```
>>> errorbar(subwave, subflux, suberror, fmt='.k', hold=True)
<matplotlib.lines.Line2D instance at 0x046EBEE0>, <a list of 204 Line2D errorbar objects>
```

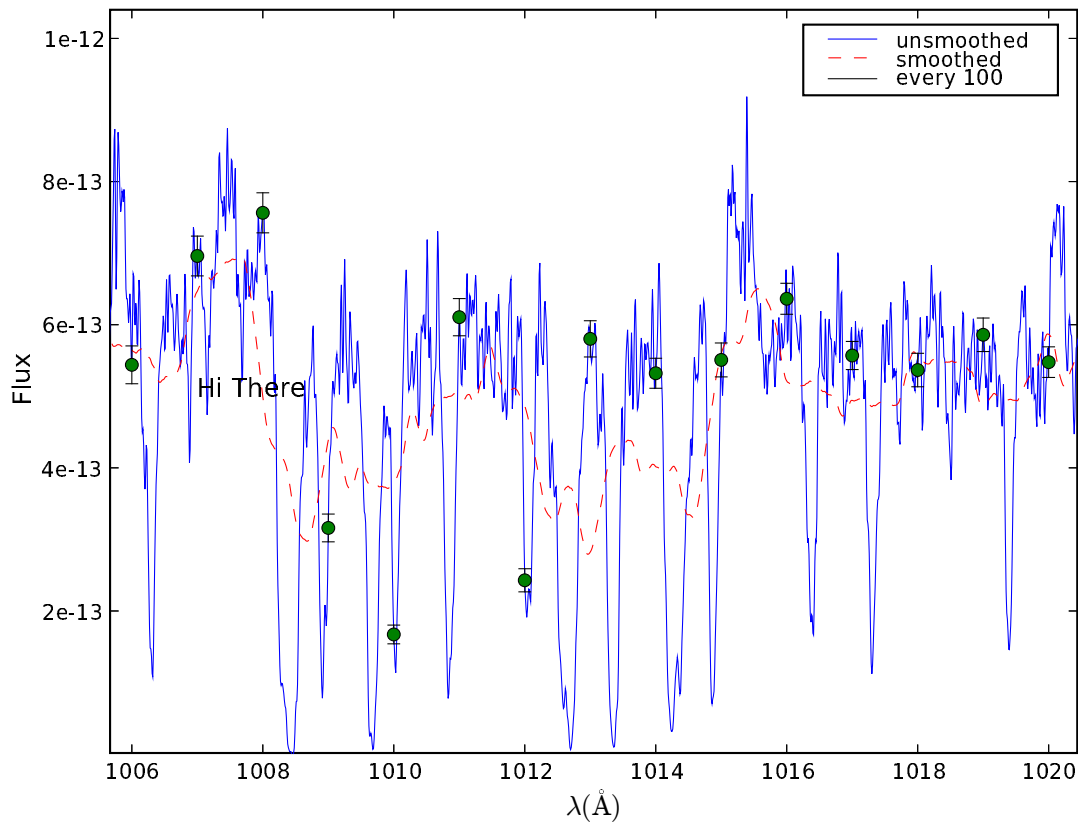
5.4.4 Legends and annotation

Adding legends is simple:

```
>>> legend(('unsmoothed', 'smoothed', 'every 100'))
<matplotlib.legend.Legend instance at 0x04978D28>
```

As is adding arbitrary text.

```
>>> text(1007., 0.5e-12, 'Hi There')
<matplotlib.text.Text instance at 0x04B27328>
```



5.4.5 Saving and printing plots

Matplotlib uses a very different style from IDL regarding how different devices are handled. Underneath the hood, matplotlib saves all the information to make the plot; as a result, it is simple to regenerate the plot for other devices without having to regenerate the plotting commands themselves (it's also why the figures can be interactive). The following saves the figure to a .png and postscript file.

```
>>> savefig('fuse.png')
>>> savefig('fuse.ps')
```

5.5 A little background on Python sequences

Python has a few, powerful built-in “sequence” data types that are widely used. You have already encountered 3 of them: strings, lists and tuples.

5.5.1 Strings

Strings are so ubiquitous, that it may seem strange treating them as a special data structure. That they share much with the other two (and arrays) will be come clearer soon. First we’ll note here that there are several ways to define literal strings. One can define them in the ordinary sense using either single quotes (') or double quotes ("). Furthermore, one can define multiline strings using triple single or double quotes to start and end the string. For example:

```
>>> s = '''This is an example
of a multi-line string that
goes on and on and on.'''
>>> s
'This is an example\nof a string that\ngo
es on and on\nand on'
>>> print s
This is an example
of a string that
goes on and on
and on
```

As with C, one uses the backslash character in combination with others to denote special characters. The most common is `\n` for new line (which means one uses `\\` to indicate a single `\` is desired). See the Python documentation for the full set. For certain cases (like regular expressions or MS Windows path names), it is more desirable to avoid special interpretation of backslashes. This is accomplished by prefacing the string with an `r` (for ‘raw’ mode):

```
>>> print 'two lines \n in this example'
two lines
in this example
>>> print r'but not \n this one'
but not \n this one
```

Strings are full fledged Python objects with many useful methods for manipulating them. The following is a brief list of all those available with a few examples illustrating their use. Details on their use can be found in the Python library documentation, the references in Appendix A or any introductory Python book. Note that optional arguments are enclosed in square brackets. For a given string `s`:

- `s.capitalize()`: capitalize first character
- `s.center(width [,fillchar])`: return string centered in string of width specified with optional fill char
- `s.count(sub [,start [,end]])`: return number of occurrences of substring
- `s.decode(encoding [,errors])`: see documentation
- `s.encode(encoding [,errors])`: see documentation
- `s.endswith(suffix [,start [,end]])`: True if string ends with suffix
- `s.expandtabs(tabsize)`: defaults to 8 spaces
- `s.find(substring [,start [,end]])`: returns position of substring found
- `s.index(substring [,start [,end]])`: like find but raises exception on failure

`s.isalnum()`: true if all characters are alphanumeric
`s.isdigit()`: true if all characters are digits
`s.islower()`: true if all characters are lowercase
`s.isspace()`: true if all characters are whitespace
`s.istitle()`: true if titlecased
`s.isupper()`: true if all characters are uppercase
`s.join(seq)`: use string to join strings in the seq (note, this is a method of the string used to join the strings in seq, not the sequence of strings!)
`s.ljust(width [,fillchar])`: return string left justified within string of specified width
`s.lower()`: convert to lower case
`s.lstrip([chars])`: strip leading whitespace (and optional characters specified)
`s.replace(old, new [,count])`: substitute old substring with new string
`s.rfind(substring [,start [,end]])`: return position of rightmost match of substring
`s.rindex(substring [,start [,end]])`: like rfind but raises exception on failure
`s.rjust(width [,fillchar])`: like ljust but right justified instead
`s.rsplit([sep [,maxsplit]])`: similar to split, see documentation
`s.split([sep [,maxsplit]])`: return list of words delimited by whitespace (or optional sep string)
`s.splitlines([keepends])`: returns list of lines within string
`s.startswith(prefix [,start [,end]])`: true if string begins with prefix
`s.strip([chars])`: strip leading and trailing whitespace
`s.swapcase()`: switch lower to upper case and visa versa
`s.title()`: return title-cased version
`s.translate(table [,deletechars])`: maps characters using table
`s.upper()`: convert to upper case
`s.zfill(width)`: left fill string with zeros to given width

```

>>> 'hello world'.upper()
'HELLO WORLD'
>>> s = 'hello world'
>>> s.find('world')
6
>>> s.endswith('ld')
True
>>> s.split()
['hello', 'world']

```

5.5.2 Lists

Think of lists as one-dimensional arrays that can contain anything for its elements. Unlike arrays, the elements of a list can be different kinds of objects (including other lists) and the list can change in size. Lists are created using simple brackets, e.g.:

```
>>> mylist = [1, 'hello', [2,3]]
```

This particular list contains 3 objects, the integer value 1, the string 'hello' and the list [2, 3].

Empty lists are permitted:

```
>>> mylist = []
```

Lists also have several methods:

`l.append(x)`: adds *x* to the end of the list

`l.extend(x)`: concatenate the list *x* to the list

`l.count(x)`: return number of elements equal to *x*

`l.index(x [,i [,j]])`: return location of first item equal to *x*

`l.insert(i, x)`: insert *x* at *i*th position

`l.pop([i])`: return last item [or *i*th item] and remove it from the list

`l.remove(x)`: remove first occurrence of *x* from list

`l.reverse()`: reverse elements in place

`l.sort([cmp [,key [,reverse]])`: sort the list in place (how sorts on disparate types are handled are described in the documentation)

5.5.3 Tuples

One can view tuples as just like lists in some respects. They are created from lists of items within a pair of parentheses. For example:

```
>>> mytuple = (1, 'hello', [2,3])
```

Because parentheses are also used in expressions, there is the odd case of creating a tuple with only one element:

```
>>> mytuple = (2) # not a tuple!
```

doesn't work since (2) is evaluated as the integer 2 instead of a tuple. For single element tuples it is necessary to follow the element with a comma:

```
>>> mytuple = (2,) # this is a tuple
```

Likewise, empty tuples are permitted:

```
>>> mytuple = ()
```

If tuples are a lot like lists why are they needed? They differ in one important characteristic (why this is needed won't be explained here, just take our word for it). They cannot be changed once created; they are called immutable. Once that "list" of items is identified, that list remains unchanged (if the list contains mutable things like lists, it is possible to change the contents of mutable things within a tuple, but you can't remove that mutable item from the tuple). Tuples have no standard methods.

5.5.4 Standard operations on sequences

Sequences may be indexed and sliced just like arrays:

```
>>> s[0]
'h'
>>> mylist2 = mylist[1:]
>>> mylist2
['hello', [2, 3]]
```

Note that unlike arrays, slices produce a new copy.

Likewise, index and slice assignment are permitted for lists (but not for tuples or strings, which are also immutable)

```
>>> mylist[0] = 99
>>> mylist
[99, 'hello', [2, 3]]
>>> mylist2
[1, 'hello', [2,3]] # note change doesn't appear on copy
>>> mylist[1:2] = [2,3,4]
>>> mylist
[1, 2, 3, 4, [2, 3]]
```

Note that unlike arrays, slices may be assigned a different sized element. The list is suitably resized.

There are many built-in functions that work with sequences. An important one is `len()` which returns the length of the sequence. E.g,

```
>>> len(s)
11
```

This function works on arrays as well (arrays are also sequences), but it will only return the length of the next dimension, not the total size:

```
>>> x = array([[1,2],[3,4]])
>>> print x
[[1 2]
 [3 4]]
>>> len(x)
2
```

For strings, lists and tuples, adding them concatenates them, multiplying them by an integer is equivalent to adding them that many times. All these operations result in new strings, lists, and tuples.

```
>>> 'hello '+'world'
'hello world'
>>> [1,2,3]+[4,5]
[1,2,3,4,5]
>>> 5*'hello '
'hello hello hello hello hello '
```

5.5.5 Dictionaries

Lists, strings and tuples are probably somewhat familiar to most of you. They look a bit like arrays, in that they have a certain number of elements in a sequence, and you can refer to each element of the sequence by using its index. Lists and tuples behave very much like arrays of pointers, where the pointers can point to integers, floating point values, strings, lists, etc. The methods allow one to do the kinds of things you need

to do to arrays; insert/delete elements, replace elements of one type with another, count them, iterate over them, access them sequentially or directly, etc.

Dictionaries are different. Dictionaries define a mapping between a key and a value. The key can be either a string, an integer, a floating point number or a tuple (technically, it must be immutable, or unchangable), but not a list, dictionary, array or other mutable objects, while the value has no limitations. So, here's a dictionary:

```
>>> thisdict = {'a':26.7, 1:['random string', 66.4], -6.3:''}
```

As dictionaries go, this one is pretty useless. There's a key whose name is the string 'a', with the floating point value of 26.7. The second key is the integer 1, and its value is the list containing a string and a floating point value. The third key is the floating point number -6.3, with the empty string as its value. Dictionaries are examples of a mapping data type, or associative array. The order of the key/value pairs in the dictionary is not related to the order in which the entries were accumulated; the only thing that matters is the association between the key and the value. So dictionaries are great for, for example, holding IRAF parameter/value sets, associating numbers with filter names, and passing keyword/value pairs to functions. Like lists, they have a number of built-in methods:

`d.copy()`: Returns a shallow copy of the dictionary (subtleties of copies will be addressed in tutorial 4)

`d.has_key(k)`: Returns True if key *k* is in *d*, otherwise False

`d.items()`: Returns a list of all the key/value pairs

`d.keys()`: Returns a list of all the keys

`d.values()`: Returns a list of all the values

`d.iteritems()`: Returns an iterator on all items

`d.iterkeys()`: Returns an iterator on all keys

`d.itervalues()`: Returns an iterator on all values

`d.get(k [,x])`: Returns *d[k]* if *k* is in *d*, otherwise *x*

`d.clear()`: Removes all items

`d.update(D)`: For each *k* in dictionary *D*, sets (or adds) *k* of dictionary = *D[k]*

`d.setdefault(k [,x])`: Returns value of *k* if *k* is in dictionary, otherwise creates key *k*, sets value to *x*, and returns *x*

`d.popitem()`: Removes and returns an arbitrary item

So, for example, we could store the ACS photometric zeropoints in a dictionary where the key is the name of the filter:

```
>>> zeropoints = {'F435W':25.779, 'F475W':26.168, 'F502N':22.352, 'F550M':24.867,
'F555W':25.724, 'F606W':26.398, 'F625W':25.731, 'F658N':22.365, 'F660N':21.389,
'F775W':25.256, 'F814W':25.501, 'F850LP':24.326, 'F892N':21.865}
>>> filter = hdr['filter1']
>>> if filter.find('CLEAR') != -1: filter = hdr['filter2']
>>> zp = zeropoints[filter]
```

Dictionaries are very powerful; if your programs do not use them regularly, you are likely doing something very wrong.

The power afforded by dictionaries, lists, tuples and strings and their built-in methods is one of the great strengths of Python.

5.5.6 A section about nothing

Python uses a special value to represent a null value called `None`. Functions that don't return a value actually return `None`. At the interactive prompt, a `None` value is not printed (but a `print None` will show its presence).

5.6 More on plotting

It is impossible in a short tutorial to cover all the aspects of plotting. The following will attempt to give a broad brush outline of matplotlib terminology, what functionality is available, and show a few examples.

5.6.1 matplotlib terminology, configuration and modes of usage

The “whole” area that matplotlib uses for plotting is called a figure. Matplotlib supports multiple figures at the same time. Interactively, a figure corresponds to a window. A plot (a box area with axes and data points, ticks, title, and labels...) is called an axes object. There may be many of these in a figure. Underneath, matplotlib has a very object-oriented framework. It's possible to do quite a bit without knowing the details of it, but the most intricate or elaborate plots most likely will require some direct manipulation of these objects. For the most part this tutorial will avoid these but a few examples will be shown of this usage.

While there may be many figures, and many axes on a figure, the matplotlib functional interface (i.e, `pylab`) has the concept of current figures, axes and images. It is to these that commands that operate on figures, axes, or images apply to. Typically most plots generate each in turn and so it usually isn't necessary to change the current figure, axes, or image except by the usual method of creating a new one. There are ways to change the current figure, axes, or image to a previous one. These will be covered in a later tutorial.

Matplotlib works with many “backends” which is another term for windowing systems or plotting formats. We recommend (for now anyway) using the standard, if not quite as snazzy, Tkinter windows. These are compatible with PyRAF graphics so that matplotlib can be used in the same session as PyRAF if you use the TkAgg backend.

Matplotlib has a very large number of configuration options. Some of these deal with backend defaults, some with display conventions, and default plotting styles (linewidth, color, background, etc.). The configuration file, `.matplotlibrc`, is a simple ascii file and for the most part, most of the settings are obvious in how they are to be set. Note that usage for interactive plotting requires a few changes to the standard `.matplotlibrc` file as downloaded (we have changed the defaults for our standard installation at STScI). A copy of this modified file may be obtained from <http://stsdas.stsci.edu/python/.matplotlibrc>. Matplotlib looks for this file in a number of locations including the current directory. There is an environmental variable that may be set to indicate where the file may be found if yours is not in a standard location.

Some of the complexity of matplotlib reflects the many kinds of usages it can be applied to. Plots generated in script mode generally have interactive mode disabled to prevent needless regenerations of plots. In such usage, one must explicitly ask for a plot to be rendered with the `show()` command. In interactive mode, one must avoid the `show` command otherwise it starts up a GUI window that will prevent input from being typed at the interactive command line. Using the standard Python interpreter, the only backend that supports interactive mode is TkAgg. This is due to the fact most windowing systems require an event loop to be running that conflicts with the Python interpreter input loop (Tkinter is special in that the Python interpreter makes special provisions for checking Tk events thus no event loop must be run). IPython has been developed to support running all the backends while accepting commands. Do not expect to be able to use a matplotlib backend while using a different windowing system within Python. Generally speaking, different windowing frameworks cannot coexist within the same process.

Since matplotlib takes its heritage from Matlab, it tends toward using more functions to build a plot rather than many keyword arguments. Nevertheless, there are many plot parameters that may be set through keyword arguments.

The `axes` command allows arbitrary placement of plots within a figure (even allowing plots to be inset within others). For cases where one has a regular grid of plots (say 2x2) the `subplot` command is used to place these within the figure in a convenient way. See one of the examples later for its use.

Generally, matplotlib doesn't try to be too clever about layout. It has general rules for how much spaces is needed for tick labels and other plot titles and labels. If you have text that requires more space than that,

It's up to you to replot with suitable adjustments to the parameters.

Because of the way that matplotlib renders interactive graphics (by drawing to internal memory and then moving to a display window), it is slow to display over networks (impossible over dial-ups, slow over broadband; gigabit networks are quite usable however)

5.6.2 **matplot functions**

The following lists most of the functions available within pylab for quick perusal followed by several examples.

basic plot types (with associated modifying functions)

`bar`: bar charts
`barh`: horizontal bar charts
`boxplot`: box and whisker plots
`contour`:
`contourf`: filled contours
`errorbar`: errorbar plot
`hist`: histogram plot
`imshow`: display image within axes boundaries (resamples image)
`loglog`: log log plot
`plot`: x, y plots
`pie`
`polar`
`quiver`: vector field plot
`scatter`
`semilogx`: log x, linear y, x y plot
`semilogy`: linear x, log y, x y plot
`stem`
`spy`: plot sparsity pattern using markers
`spy2`: plot sparsity pattern using image

plot decorators and modifiers

`axhline`: plot horizontal line across axes
`axvline`: plot vertical line across axes
`axhspan`: plot horizontal bar across axes
`axvspan`: plot vertical bar across axes
`clabel`: label contour lines
`clim`: adjust color limits of current image
`grid`: set whether grids are visible
`legend`: add legend to current axes
`rgrids`: customize the radial grids and labels for polar plots
`table`: add table to axes
`text`: add text to axes
`thetagrids`: for polar plots
`title`: add title to axes

xlabel: add x axes label
ylabel: add y axes label
xlim: set/get x axes limits
ylim: set/get y axes limits
xticks: set/get x ticks
yticks: set/get y ticks

figure functions

colorbar: add colorbar to current figure
figimage: display unresampled image in figure
figlegend: display legend for figure
figtext: add text to figure

object creation/modification/mode/info functions

axes: create axes object on current figure
cla: clear current axes
clf: clear current figure
close: close a figure window
delaxes: delete axes object from the current figure
draw: force a redraw of the current figure
gca: get the current axes object
gcf: get the current figure
gci: get the current image
hold: set the hold state (overdraw or clear?)
ioff: set interactive mode off
ion: set interactive mode on
isinteractive: test for interactive mode
ishold: test for hold mode
rc: control the default parameters
subplot: create an axes within a grid of axes

color table functions

autumn, bone, cool, copper, flag, gray, hot, hsv, pink, prism, spring, summer, winter

5.7 Plotting mini-Cookbook

5.7.1 customizing standard plots

The two tables below list the properties of data and text properties and information about what values they can take. The variants shown in parentheses indicate acceptable abbreviations when used as keywords.

Data properties

Property	Value
alpha	Alpha transparency (between 0. and 1., inclusive)
antialiased (aa)	Use antialiased rendering (True or False)
color (c)	Style 1: 'b' -> blue 'g' -> green 'r' -> red 'c' -> cyan 'm' -> magenta 'y' -> yellow 'k' -> black 'w' -> white Style 2: standard color string, eg. 'yellow', 'wheat' Style 3: grayscale intensity (between 0. and 1., inclusive) Style 4: RGB hex color triple, eg. #2F4F4F Style 5: RGB tuple, e.g., (0.18, 0.31, 0.31), all values between 0. and 1.)
data_clipping	Clip data before plotting (if the great majority of points will fall outside the plot window this may be much faster); True or False
label	A string optionally used for legend
linestyle (ls)	One of '-' (dashed), ':' (dotted), '-.' (dashed dot), '-' solid
linewidth (lw)	width of line in points (nonzero float value)
marker	symbol: 'o' -> circle '^', 'v', '<', '>' triangles: up, down, left, right respectively 's' -> square '+' -> plus 'x' -> cross 'D' -> diamond 'd' -> thin diamond '1', '2', '3', '4' tripods: down, up, left, right 'h' -> hexagon 'p' -> pentagon ' ' -> vertical line '_' -> horizontal line 'steps' (keyword arg only)
markeredgewidth (mew)	width in points (nonzero float value)
markeredgecolor (mec)	color value
markerfacecolor (mef)	color value
markersize (ms)	size in points (nonzero float value)

The following describes text attributes (those shared with lines are not detailed)

Property	Value
alpha, color	As with Lines
family	font family, eg 'sans-serif', 'cursive', 'fantasy'
fontangle	the font slant, 'normal', 'italic', 'oblique'
horizontalalignment	'left', 'right', 'center'
verticalalignment	'top', 'bottom', 'center'
multialignment	'left', 'right', 'center' (only for multiline strings)
name	font name, eg. 'Sans', 'Courier', 'Helvetica'
position	x, y position
variant	font variant, eg. 'normal', 'small-caps'
rotation	angle in degrees for text orientation
size	size in points
style	'normal', 'italic', or 'oblique'
text	the text string itself
weight	e.g 'normal', 'bold', 'heavy', 'light'

These two sets of properties are the most ubiquitous. Others tend to be specialized to a specific task or function. The following illustrates with some examples (plots are not shown) starting with the ways of specifying red for a plotline

```
>>> x = arange(100.)
>>> y = (x/100.)**2
>>> plot(x,y,'r')
>>> plot(x,y, c='red')
>>> plot(x,y, color='#ff0000')
>>> lines = plot(x,y) # lines is a list of objects, each has set methods
                        # for each property
>>> set(lines, 'color', (1.,0,0)) # or
>>> lines[0].set_color('red') ; draw() # object manipulation example
>>> plot(y[::10], 'g>:', markersize=20) # every 10 points with large green triangles and
                        # dotted line
```

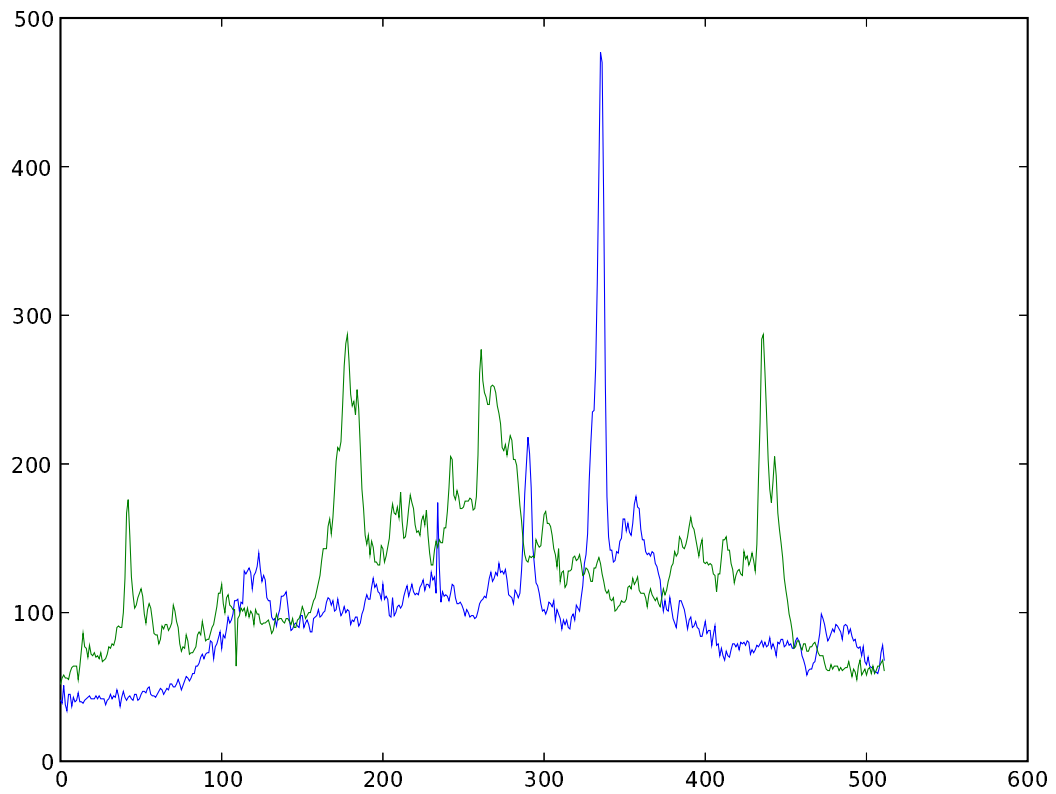
And more examples specifying text:

```
>>> textobj = xlabel('hello', color='red', ha='right')
>>> set(textobj, 'color', 'wheat') # change color
>>> set(textobj, 'size', 5) # change size
```

5.7.2 implot example

You can duplicate simple use of the IRAF task `implot` like this:

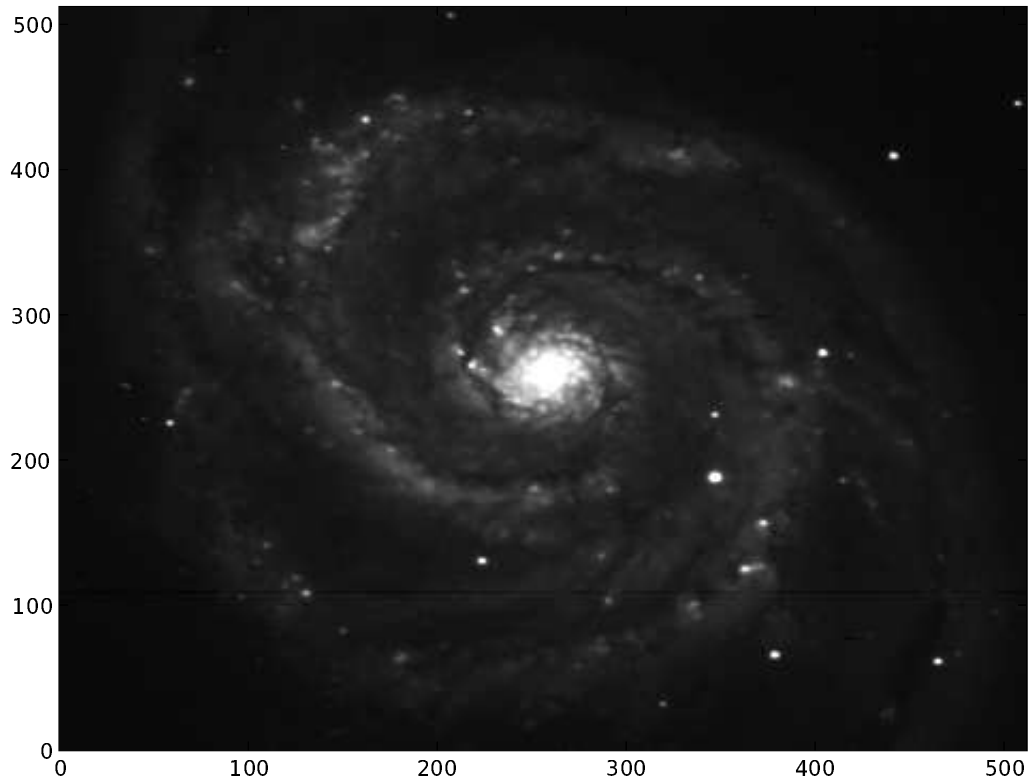
```
>>> pixdata = pyfits.getdata('pix.fits')
>>> plot(pixdata[100], hold=False) # plots row 101
>>> plot(pixdata[:,200], hold=True) #overplots col 201
```



5.7.3 imshow example

Images can be displayed both using `numdisplay`, which was introduced in the last tutorial and the `matplotlib` `imshow` command:

```
>>> import numdisplay
>>> numdisplay.open()
>>> numdisplay.display(pixdata,z1=0,z2=1000)
>>> clf() # clears the current figure
>>> imshow(pixdata,vmin=0,vmax=1000)
>>> gray() # loads a greyscale color table
```



The default type of display in Matplotlib “out of the box” has the Y coordinate increasing from top to bottom. This behavior can be overridden by changing a line in the `.matplotlibrc` file:

```
image.origin : lower
```

The `.matplotlibrc` that is the default on STScI unix systems (Solaris, Linux and Mac) has this already set up. If you are using Windows, you will need to get the STScI default `.matplotlibrc` from one of the Unix systems, or from the web.

`imshow` will resample the data to fit into the figure using a defined interpolation scheme. The default is set by the `image.interpolation` parameter in the `.matplotlibrc` file (`bilinear` on STScI systems), but this can be set to one of `bicubic`, `bilinear`, `blackman100`, `blackman256`, `blackman64`, `nearest`, `sinc144`, `sinc256`, `sinc64`, `spline16` and `spline36`. Most astronomers are used to blocky pixels that come from using the nearest pixel; so we can get this at run-time by doing

```
>>> imshow(pixdata, vmin=0, vmax=1000, interpolation='nearest')
```

5.7.4 figimage

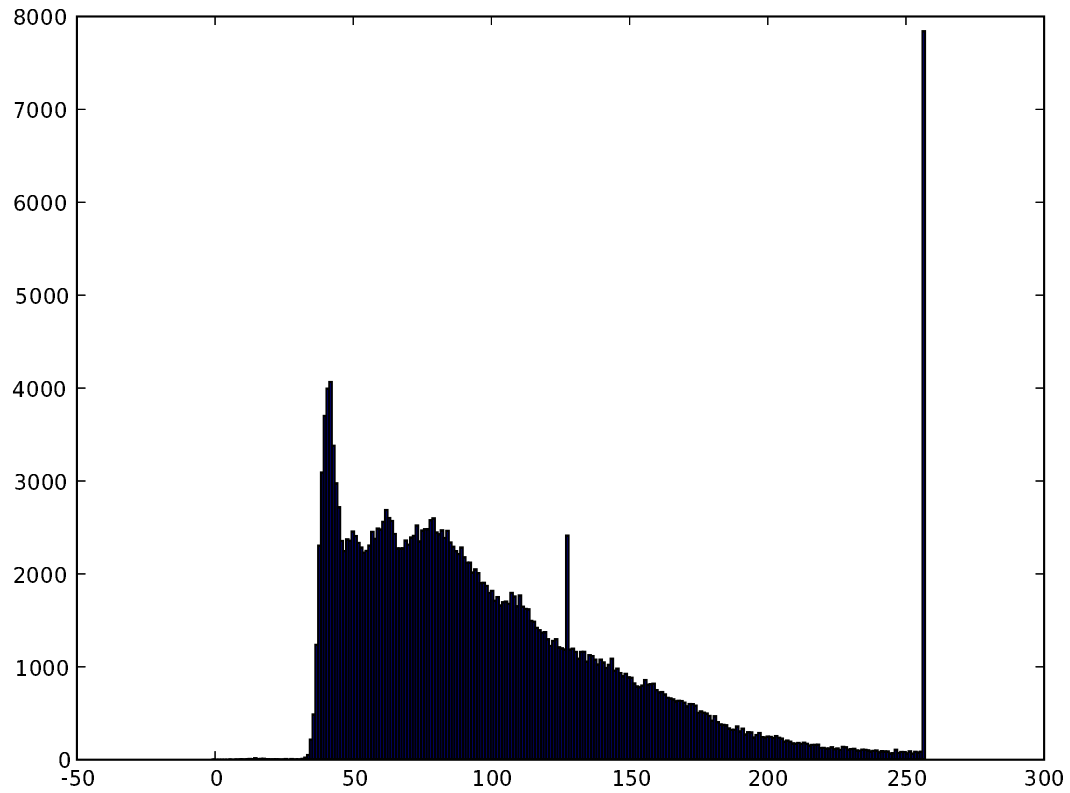
`figimage` is like `imshow`, except no spatial resampling is performed. It also allows displaying RGB or RGB-alpha images. This function behaves more like the IDL TVSCL command. See the manual for more details.

5.7.5 histogram example

Histograms can be plotted using the `hist` command.

```
>>> pixdata[pixdata>256] = 256
>>> hist(pixdata,bins=256)
```

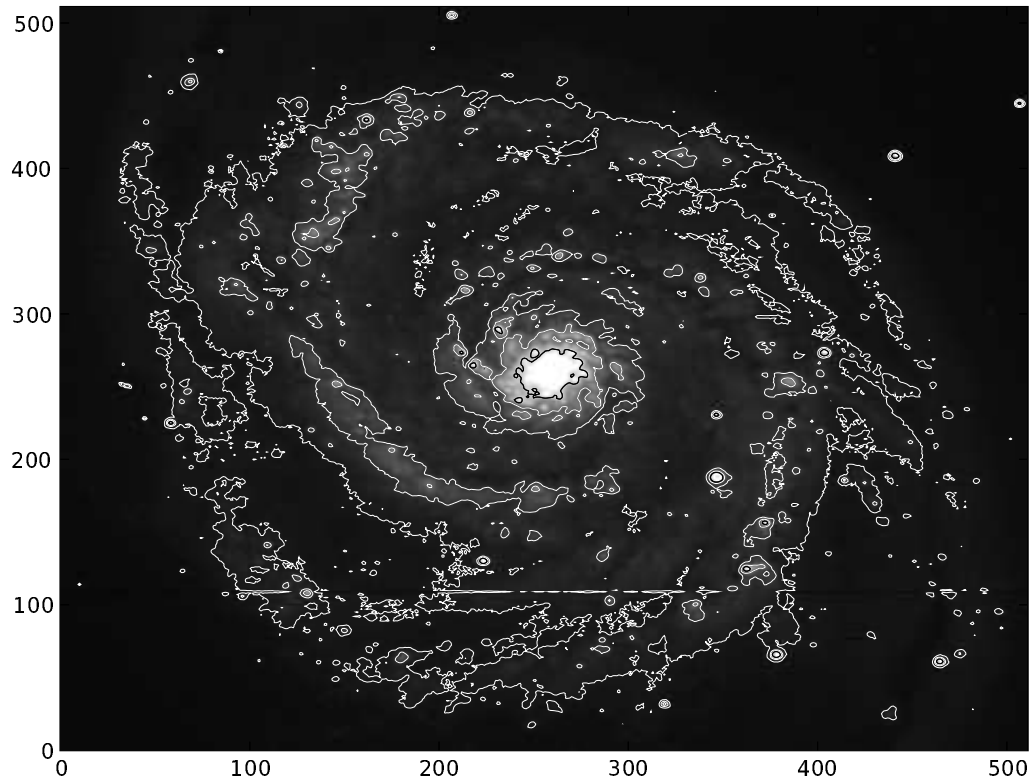
Duplicating the functionality of, for example, the IRAF `imhistogram` task will take a bit more work.



5.7.6 contour example

To overplot green contours at levels of 100, 200, 400 and 800, we can do:

```
>>> levels = [100,200,400,800]
>>> imshow(pixdata,vmin=0,vmax=1000,origin='lower')
>>> contour(pixdata,levels,colors=[1., 1., 1., 0.]
```



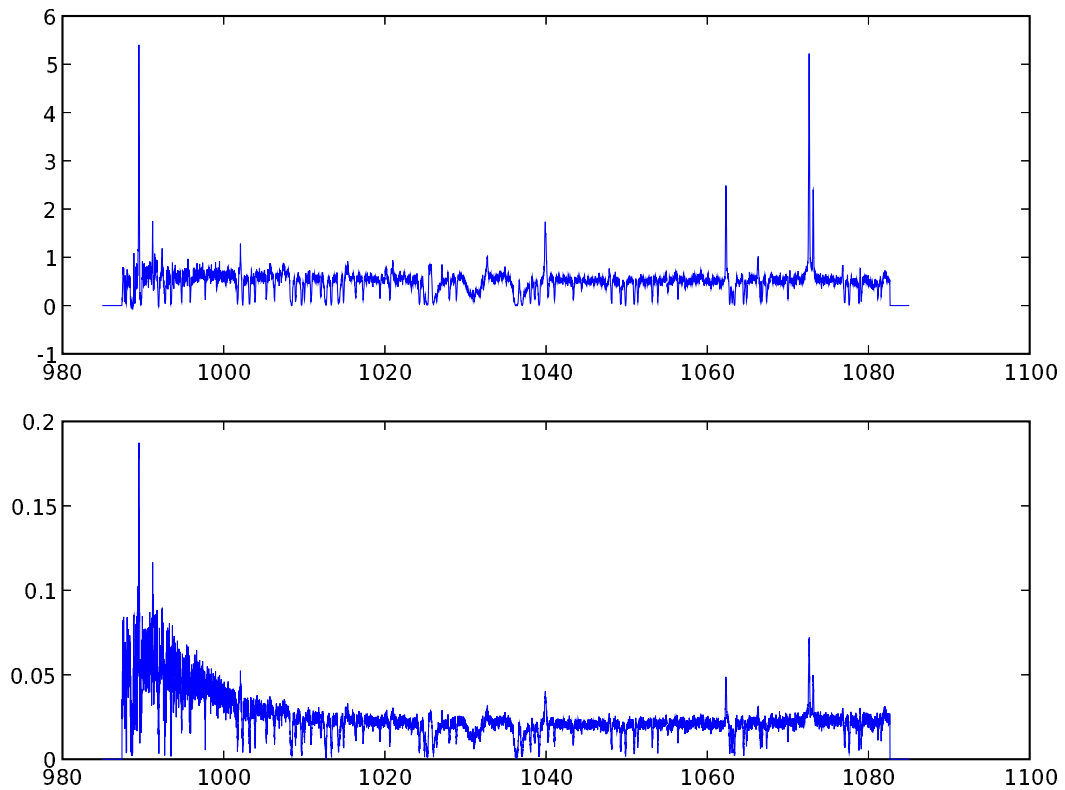
5.7.7 subplot example

You aren't limited to having one plot per page. The `subplot` command will divide the page into a user-specified number of plots:

```
>>> err = err*1.0e12
>>> flux = flux*1.0e12
```

There's a bug in the `axes` code of Matplotlib that causes an exception when the upper and lower limits of the plot are too close to each other, so we multiply the data by `1.0e12` to guard against this.

```
>>> subplot(211)                # Divide area into 2 vertically, 1
                                # horizontally and select the first plot
>>> plot(wavelength,flux)
>>> subplot(212)                # Select second plot
>>> plot(wavelength,error)
```



5.7.8 readcursor example

Finally, it is possible to set up functions to interact with a plot using the event handling capabilities of `Matplotlib`. Like many other GUIs, `Matplotlib` provides an interface to the underlying GUI event handling mechanism. This is achieved using a callback function that is activated when a certain prescribed action is performed. The prescribed action could be a mouse button click, key press or mouse movement. You can set up a handler function to handle the event by registering the event you want to detect, and connecting your callback function with the required event. This is getting a little bit ahead of ourselves here, since we weren't going to explain functions until the fourth tutorial, but it's impossible to explain event handling without it...

This is best explained using an example. Say you want to print the X and Y coordinates when you click the mouse in an image. Start by displaying the image in an `imshow()` window:

```
>>> imshow(pixdata, vmin=0, vmax=200)
```

Then set up a handler to print out the X and Y coordinates. This will be run every time the mouse button is clicked, until the "listener" function is killed.

```
>>> def clicker(event):
...     if event.inaxes:
...         print event.xdata, event.ydata
...
>>>
```

This is a very simple handler: it just checks whether the cursor is inside the figure (`if event.inaxes`), and if it is, it prints out the X and Y coordinates in data coordinates (i.e. in the coordinates as specified by the axes). If the cursor is outside the figure, nothing is done.

Now we set up a “listener” by connecting the event we are looking for (`'button_press_event'`) with the function we will call (`clicker`).

```
>>> cid = connect('button_press_event', clicker)
```

The connect function returns a “connect id”, which we will use when we want to kill the listener.

Now, when we press the mouse button when we are in the figure, we get the coordinates printed out on our screen. This works even if we zoom the image, since we have requested the coordinates in data space, so matplotlib takes care of determining the X and Y coordinates correctly for us. When we’re done, we can just do the following:

```
>>> disconnect(cid)
```

and we are no longer listening for button presses.

There are more details in the matplotlib manual.

5.8 Exercises

1. Using only Python tools, open the FITS file `fuse.fits`, extract the flux and error columns from the table, and plot these against each other using `plot()`. Scale the flux and error by $1.0e12$ before plotting.
2. Smooth `pix.fits` with a 31×31 boxcar filter (see first tutorial) and make a contour plot at 90, 70, 50, and 30, and 10% of the peak value of the smoothed image.
3. Repeat 2, except before generating a contour plot display the unsmoothed image underneath.
4. Change the colortable to gray and label your favorite feature with a mathematical expression
5. Save the result as a postscript file and view it in your favorite postscript viewer
6. Extra credit: parroting the readcursor example, write a short callback function to display a cross cut plot of a displayed image on a left mouse button click, and a vertical cut plot on a right button click (overplotting the image). [Hint: this needs use of `subplot(111, Frameon=False)` to get overplotting on images to work]

6 Tutorial 3: More advanced topics in PyFITS, numarray and IPython

6.1 IPython

IPython is a souped up interactive environment for Python. It adds many special features to make interactive analysis sessions easier and more productive. These features include:

- easy invocation of common shell commands with shell-like syntax (e.g., `ls`, `cd`, etc.)
- TAB completion
- Easy ways to find out information about modules, functions and objects.
- Numbered input/output prompts with command history
- User-extensible magic commands
- Alias facility for defining system aliases
- Complete system shell access (through `!` and `!!` prefixes)
- Session logging and restoring (through playback of logged commands)
- Background execution of Python commands in a separate thread
- Automatic indentation
- Macro system
- Auto parenthesis and auto quoting
- Flexible configuration system allowing for multiple configurations
- Easy reimportation (useful for script and module development)
- Easy debugger access
- Syntax highlighting and editor invocation, easier to interpret tracebacks
- Profiler support

This tutorial will only touch a subset of its extensive features (IPython has its own, very complete manual: see Appendix A for instructions on how to obtain it). When started (simply by typing `ipython`) the most apparent difference from the standard Python interpreter is the prompt. Each line entered by the user is numbered. Commands can be recalled through the standard up and down arrow keystrokes (if the readline library is installed) as well as by number (shown later).

6.1.1 Obtaining information about Python objects and functions

IPython provides a few very handy ways of finding out information about modules, functions, and objects. Some examples:

```
In [10]: x = [1, 2] # create a short list
```

Typing `?` before or after a name (module, variable, etc.) gives information about that module or object including documentation included in the source code for that module or object (the “docstring”, discussed later).

```

In [11]: x?
Type:      list
Base Class: <type 'list'>
String Form: [1, 2]
Namespace?: Interactive
Length:    2
Docstring:
    list() -> new list
    list(sequence) -> new list initialized from sequence's items

```

This shows a little info about the object.

If one types a TAB character after the object or module name with a period, IPython will list all the possible contents of that module or object (i.e., its attributes or methods). One can then look at any documentation available for these items using the ? feature. E.g.,

```

In [12]: x.<TAB>
x.__add__      x.__iadd__      x.__setattr__
x.__class__    x.__imul__      x.__setitem__
x.__contains__ x.__init__      x.__setslice__
x.__delattr__  x.__iter__      x.__str__
x.__delitem__  x.__le__        x.append
x.__delslice__ x.__len__       x.count
x.__doc__      x.__lt__        x.extend
x.__eq__       x.__mul__       x.index
x.__ge__       x.__ne__        x.insert
x.__getattr__  x.__new__       x.pop
x.__getitem__  x.__reduce__    x.remove
x.__getslice__ x.__reduce_ex__ x.reverse
x.__gt__       x.__repr__      x.sort
x.__hash__     x.__rmul__
In [13]: x.append?
Type:          builtin_function_or_method
Base Class:    <type 'builtin_function_or_method'>
String Form:   <built-in method append of list object at 0x15bcff0>
Namespace:     Interactive
Docstring:
    L.append(object) -- append object to end

```

This lists all the attributes and methods for lists. Generally you should ignore any names that begin with underscore (the meaning of all these will be explained in the next tutorial). The next inquiry shows the docstring info for the list append method.

These techniques can be used to inspect what's available within any Python object or module. One caution: objects that use more sophisticated means of handling attributes and methods will generally not show these by these inspection tools. So just because you don't see something doesn't mean it doesn't exist. A good example is the data attribute for pyfits objects (something that will be covered in this tutorial). There is such an attribute, but it won't be seen through the TAB completion mechanism here.

6.1.2 Access to the OS shell

Any system shell command may be executed by prepending it with '!'. E.g.,

```

In [14]: !ls
newfile.fits  seq.fits  pix.fits

```

6.1.3 Magic commands

IPython has what it calls magic commands. These are special commands not normally part of Python. The following lists the default set. Users may define their own magic commands.

Magic Command	Action
Exit	Exit IPython without confirmation
Pprint	Toggle pretty printing on/off
Quit	same as Exit
alias	Define an alias for a system command
autocall	Make functions callable without typing parenthesis (default)
autoindent	Toggle autoindent on/off (default on)
automagic	Make magic functions callable without having to type '%' (default)
bg	Run a job in the background, in a separate thread
bookmark	Manage bookmark system (for directories)
cd	Change directories
color_info	Toggle color syntax highlighting (default on)
colors	switch color scheme
config	Show IPython's internal configuration
dhist	Print history of visited directories
dirs	Return current directory stack
ed	Alias to edit
edit	Bring up editor and execute the resulting code
env	List environmental variables
hist	Print input history, most recent last
logoff	Stop logging
logon	Restart logging
logstart	Start logging
logstate	Print the status of the logging system
lsmagic	List currently available magic functions
macro	Define set of input lines as macro
magic	Print info about magic function system
p	Alias for Python print function
page	Pretty print the object and display it through a pager
pdb	Control the calling of the pdb interactive debugger
pdef	Print the definition header of any callable object
pdoc	Print the docstring for an object
pfile	Print the file where an object is defined
pinfo	Provide detailed information about an object
popd	Change to a directory popped off the directory stack
profile	Print currently active IPython profile
prun	Run a statement through the Python code profiler
psource	Print source code for object
pushd	Push current directory on stack and change directory
pwd	Return current working directory
r	Repeat previous input
rehash	Update the alias table with all entries in \$PATH
rehashx	Update the alias table with all executables in \$PATH
reset	Reset the namespace by removing all names defined by user
run	Run the named file inside IPython as a program

runlog	Run files as logs
save	Save a set of lines to a given filename
sc	Execute a shell command and capture its output
sx	Run a shell command and capture its output
system_verbose	Toggle verbose printing of system commands on and off
time	Time execution of a Python statement or expression
unalias	Remove alias
who	Print all interactive variables with minimal formatting
who_ls	Return sorted list of all interactive variables
whos	Like who but with extra info about each variable
xmode	Switch modes for exception handlers

The use of many of these is fairly obvious; for others consult the IPython documentation on their use.

IPython has default aliases for the following system commands: `cat`, `clear`, `cp`, `less`, `mkdir`, `mv`, `rm`, `rmdir`, and many variants on `ls` (`lc`, `ldir`, `lf`, `lk`, `ll` and `lx`). The magic `alias` command allows one to change or add to the list.

IPython saves the history of previous commands from one session to the next (the manual describes how to keep different session histories).

6.1.4 Syntax shortcuts

With autocall mode on, it means that it is not necessary to provide parentheses for functions that aren't being assigned:

```
In [15]: pyfits.info 'pix.fits' # instead of pyfits.info('pix.fits')
----->pyfits.info('pix.fits')
Filename: pix.fits
No.    Name          Type          Cards  Dimensions  Format
0     PRIMARY      PrimaryHDU    71    (512, 512)  Int16
```

but note that this doesn't work:

```
In [16]: im = pyfits.getdata 'pix.fits'
-----
File "<console>", line 1
      im = pyfits.getdata 'pix.data'

SyntaxError: invalid syntax
```

If one starts a line with a comma, every argument of a function is assumed to be auto quoted. E.g.:

```
In [17]:,pyfits.info pix.fits
----->pyfits.info('pix.fits')
Filename: pix.fits
No.    Name          Type          Cards  Dimensions  Format
0     PRIMARY      PrimaryHDU    71    (512, 512)  Int16
In [18]:,print hello
----->print('hello')
hello
```

Note also that when IPython does this sort of autotransformation, it prints the command actually executed. What is saved in the log files is the proper Python syntax, not what you typed. For functions that expect strings only, this allows near shell-like syntax to be used.

6.1.5 IPython history features

Typing `Control-p` (or `Control-n`) after you typed something at the command line will match the closest previous (or next) command in the history that matches what you've typed.

Previous input commands are available for re-execution (or other manipulation) by accessing special variables. The variable `In` is effectively a list of the previous commands that can be indexed with the number that matches the prompt. Slicing works as well. The following example show how to select a set of previous lines for re-execution:

```
In[19]: print 1
1

In[20]: print 2
2

In[21]: print 3
3

In[22]: print 4
4
In [23]: exec In[19:21]+In[22]
1
2
4
```

The previous lines are also accessible through variables `_i<n>` and `_ih<n>`. E.g.,

```
In [24]: print _i22
print 4
```

The return value of statements that return a value (and thus often print or display something to the output are stored in similar variables). The most recent is in a variable `_`, the next most recent in `__`, and likewise for `___`. For example:

```
In[25]: 6+9
15
In[26]: print _
15
```

Since assignments don't return anything, they are not saved this way. Like input commands, output values are saved in `Out` and can be indexed, or the variables `_o<n>` or `_oh<n>`. This feature may use extra memory since it is saving references to things that normally would be deleted so the feature can be turned off (see the later discussion on memory usage for a better understanding of what happens).

By defining a `.ipythonrc` file in each working directory, one can customize what IPython does for a session started in that directory. Additional IPython features will be illustrated throughout the rest of the tutorials.

6.2 Python Introspection

Snake navel gazing? No, not quite. This term refers to the capabilities that Python provides for finding out information about Python objects from the objects themselves (IPython makes use of these facilities). Python has very extensive tools for doing this. Only a couple of the simplest will be mentioned here. By using the Python `dir()` function, one can examine the “contents” of various Python entities including modules, objects and classes. For example using `dir()` on a module will list variables, functions, and classes defined within the module (but not indicate what they are; in the list there is no visual way of distinguishing a function from a variable—except by use of standard naming conventions in some cases. As an example

`dir(pyfits)` lists over a hundred such items. This can be useful if you can't quite remember the name of a method or attribute. By itself, don't expect to understand everything you see (or even most). One can use `dir()` on these items in turn to find out about them.

The Python function `type()` can be used on any object to tell you what type it is. From its use you can tell if something is an integer, string, list, function, or specific object.

Every Python module, function, class and method has a provision for a special string that is available for inspection. This string is called the "docstring". If it exists, it is the `__doc__` attribute of the object. Here are a few examples:

```
>>> x = [1,2] # a simple python list
>>> print x.__doc__
list() -> new list
list(sequence) -> new list initialized from sequence's items
This shows the docstring for the list creator function.
>>> print pyfits.__doc__ # print the pyfits module docstring
A module for reading ....
....
>>> print pyfits.getdata.__doc__
...
```

With these few features it is usually possible to find out quite a bit about any object.

6.3 Saving your data

At the moment, there are no comparable functions to the IDL `save` and `restore` commands. There are ways of saving and restoring Python objects (which goes by the odd name of pickling) but its use is far from being convenient as the IDL `save` command. We are working on a command to make saving and restoring data much easier (and similar to how it is done in IDL), but given the richness of the forms that Python objects can be generated, it is not possible to ensure that all objects can be saved. The future command will allow saving of relatively simple objects (arrays, lists, dictionaries, etc)

6.4 Python loops and conditionals

6.4.1 The for statement

Like many other languages Python has a for statement. Its use is a little different than most are likely used to seeing. Here is an example:

```
>>> pets = ['dog', 'cat', 'gerbil', 'canary', 'goldfish', 'rock']
>>> for pet in pets: print 'I have a pet', pet
I have a pet dog
I have a pet cat
I have a pet gerbil
I have a pet canary
I have a pet goldfish
I have a pet rock
```

The `for pet in pets` syntax means that a loop will be performed for each item in the list `pets`. In order, the item is taken from list, assigned to the variable `pet`, which can be used within the loop. Anything that is considered a Python sequence can be used after the `in`. This includes lists, strings, tuples, arrays and anything else (including user-defined objects) that satisfy certain sequence requirements (actually, it's more general than that, iterator objects, not described here can be use used as well). In fact, the exact same approach is used for performing "counted" loops:

```
>>> for i in range(3): print i
0
```

```
1
2
```

The function `range(n)` effectively generates a list of `n` integers starting at 0 (it's the analog of `arange(n)`). It turns out that it doesn't actually generate a list that long (so worries about using up lots of memory for performing a large number of loops are unfounded).

6.4.2 Blocks of code, and indentation

The form of having the code looped over on the same line as the for statement is atypical. Normally the statements being looped over follow on separate lines. So what is used to delimit these blocks? Begin/end statements like IDL? Curly braces (`{}`) like C and Java? No, nothing other than simple indentation. When code following a statement that expect a block of code (such as for) is consistently indented more than the statement starting the block, it is considered to identify the block of code. For example:

```
>>> for i in range(3):
...     x = i*i
...     print x
...
0
1
4
```

Note that when one is in interactive mode, that using a statement that starts a block causes the interpreter to prompt with `...` and typing an empty line terminates the block (that's not necessary within scripts or programs, returning to a previous level of indentation or ending the file is sufficient). The amount of indentation is entirely up to you. It can be 1 space or 40, so long as it is consistent. Tabs may be used, but their use is not recommended. If you do use them, use only tabs for indentation. A mixture of tabs and spaces is bound to cause confusion since different tools show tabs with varying amounts of equivalent spaced. Most sensible editors allow tabs to be automatically converted to spaces with the appropriate configuration (the next tutorial will provide information about editors and available features for Python).

This aspect of Python astonishes some, but most that actually try it find it refreshing. The structure that you see is the real structure of the program. Cutting and pasting such code into different levels of indentation is not usually a problem if you use an editor that supports changing indentation for multiple lines of code.

6.4.3 Python if statements

Python if statements are simple:

```
>>> x = 0
>>> if x==0:
...     print "x equals 0"
...
x equals 0
```

The if statement allows optional `elif` and `else` clauses:

```
>>> if x==1:
...     print "x equals 1"
... elif x==0:
...     print "x equals 0"
... else:
...     print "x equals something other than 1 or 0"
...
x equals 0
```

6.4.4 What's True and what's False

Python now has explicit `True` and `False` values. But it also has standard rules for determining what is considered true and false for other data types. Numeric data types are considered false if equal to 0 and true for all other values. The Python value `None` is always considered false. Empty sequences and dictionaries are considered false (e.g., the empty string).

```
>>> if '': print 'True' # nothing will be printed
...
>>> if '0': print 'True'
...
True
```

As for user-defined objects, Python permits the code to define what is true or not. Or whether a test for truth is even permitted; `numarray` arrays may not be used as truth values:

```
>>> from numarray import *
>>> x = arange(9)
>>> if x: print 'True'
[...]
RuntimeError: An array doesn't make sense as a truth value. Use sometrue(a) or all true(a)
```

Error!

6.5 Advanced PyFITS Topics

The PyFITS functions introduced in the first tutorial are sufficient for simple manipulation of FITS files, but if you need to have more control over the handling of the FITS file, you will need to use some of the more advanced features of the module.

6.5.1 Header manipulations

Normally most people will find the means of using and updating headers described in the first tutorial sufficient for most tasks. There are more methods available for header objects that allow adding commentary keywords (`HISTORY`, `COMMENT`, or 'blank' keywords). The following show their use.

```
>>> hdr = pyfits.getheader('pix.fits')
>>> hdr.add_history('This is a test history card')
>>> hdr.add_comment('A test comment card',after='date')
>>> hdr.add_blank('',before='comment')
# To delete a keyword just use the del statement
>>> del hdr['date']
```

Note that while headers behave in a number of respects like dictionaries, they aren't dictionaries and don't support all dictionary methods.

To have complete control over header cards and their placement of header cards it is necessary to access the header as a "cardlist" object. With this representation one can use list manipulations to place each card as desired, and use card methods to control the exact contents of any card. By this interface it is possible to make the card use any string desired, even if it is not legal FITS or one of the standard conventions (to write such illegal cards into an output file requires specifically instructing the `writeto` method or function to force illegal cards into the file; see the previous section).

See the PyFITS documentation for the full set of methods and examples of use.

6.5.2 PyFITS Object Oriented Interface

The object oriented interface permits more flexible handling of FITS files. The object functionality mirrors that available from the functions. The chief advantages are:

- Files are not repeatedly opened and closed for each data access (but the objects should be explicitly closed now with the `close()` method).
- Allows creating FITS headers from scratch (i.e., not having to use an existing file's header).
- There is more flexibility in how data are accessed. One can easily determine how many extensions exist and access all the headers directly without making many `getheader` function calls.

As for the latter, it is possible to read just portions of an image from disk (rather than reading the whole image), and it is possible to open files using memory mapping as a data access mode. Consult the PyFITS User's Guide for details on how to do both of these.

Simple example:

```
>>> hdus = pyfits.open('fuse.fits') # returns an HDUList object
>>> tabhdu = hdus[1] # the table extension
>>> hdr = tabhdu.header # getting the header from the HDU
>>> tab = tabhdu.data # and the table
```

The data attribute of the HDU object contains the data regardless of the type of extension (a record array for tables, a simple array for image extensions). The `HDUList` may be indexed by number or by extension name (and optionally `EXTVER`). For example:

```
>>> hdus = pyfits.open('acs.fits')
>>> sci = hdus['sci',1] # EXTNAME=SCI, EXTVER=1
```

`HDUList`s can be written to a new file, or updated if the file was opened in update mode.

```
>>> hdus.writeto('temp.fits') # write current form to new file
>>> hdus.flush() # update opened file with current form
```

6.5.3 Controlling memory usage

Offhand, it sounds convenient to read a whole FITS file into memory as an `HDUList` in one step, but some may wonder about the demands this puts on memory. If the file is memory mapped, then that isn't a problem. However, if the default mode of using regular I/O to read FITS files is being used it turns out that only the headers are loaded. Until one actually refers to the `data` attribute the data are not read into memory (it's done with smoke and mirrors). By using this, it is possible to control what is in memory. The following illustrates how (the next tutorial unveils some of the details of how memory is handled in general in Python).

```
>>> ff = pyfits.open('nicmos.fits')
>>> sum = 0. # plan is to sum all SCI extension images
>>> for hdu in ff[1:]: # skip primary
...     if hdu.header['extname'].startswith('SCI'):
...         sum = sum + hdu.data # now data is read in
...         hdu.data = None # done with it, delete it from memory
>>> ff.close()
```

If the images are too large to comfortably process in memory, the `section` attribute can be used to read subsets of the image. An example that sums all the values in sections of an image without loading it all in memory at once:

```

>>> ff = pyfits.open('pix.fits') # returns an HDUList object
>>> hdu = ff[0] # Get first (and only) HDU
>>> sum = 0L # for keeping a running sum of all image values
>>> # now read data 64 lines at a time (although small, it illustrates the principle)
>>> for i in range(8):
...     subim = hdu.section[i*64:(i+1)*64,:]
...     sum = sum + subim.sum()
...
>>> print sum
>>> ff.close()

```

This can be done more elegantly using image iterators described later.

HDULists can be created and be manipulated entirely in memory without any reference to any input or output files.

PyFITS is also capable of handling variable-length table entries, random groups, and ascii table extensions.

6.5.4 Support for common FITS conventions

PyFITS supports the CONTINUE convention automatically on reading files that use it (this permits the use of string values longer than permitted in a standard card). To enable its usage in a header just provide string values to keywords that are longer than standard FITS allows. The HIERARCH convention is not yet supported, but will be in the near future. The INHERIT convention is not supported. It is not planned to be in the base pyfits module but some convenience functions or objects may be provided in the future to support it.

6.5.5 Noncompliant FITS data

Normally PyFITS tries to handle FITS files that don't meet the FITS standard as best it can when reading them in (and made the most reasonable interpretations possible of any ambiguities). (If users encounter problems in this regard, particularly with FITS data that is widely used, please let us know so we can adjust PyFITS to handle it). In contrast, PyFITS normally takes a very strict view toward the FITS files it writes. By default, PyFITS will not permit non-compliant FITS files to be written. When the errors in the input file have obvious corrections, PyFITS will automatically make these corrections in writing the header to a new file with either of the fix options (e.g., `output_verify='fix'` or `output_verify='silentfix'`). When there is ambiguity about what should be done, PyFITS will raise an exception. It is possible to change these behavior, i.e., to warn or raise errors on bad input files, and to permit FITS non-compliant headers to be written. Some illustrations:

```

>>> hdu = pyfits.PrimaryHDU() # create an HDU from scratch
>>> hdu.header.update('P.I.', 'Hubble') # Try creating an illegal keyword
ValueError: Illegal keyword name 'P.I.'
# force into header an illegal keyword
>>> card = pyfits.Card().fromstring("P.I.          = 'Hubble'")
>>> hdu.header.ascardlist().append(card)
>>> hdu.writeto('pi.fits') # try writing to file
Output verification result:
HDU 0:
  Card 4:
    Unfixable error: Illegal keyword name 'P.I'
[...]
  raise VerifyError
VerifyError
# now force it to file
>>> hdu.writeto('pi.fits', output_verify='ignore')

```

```

>>> hdus = pyfits.open('pi.fits')
>>> print hdus[0].header.ascardlist()
[...]
P.I.      = 'Hubble'
>>> hdus[0].header['p.i. ']
'Hubble'
>>> hdus.verify()
[repeat of previous error message]

```

The `verify()` method (and the `output_verify` keyword) has the possible values of: `'ignore'`, `'fix'`, `'silentfix'`, `'exception'` (default for output), and `'warn'`.

6.6 A quick tour of standard numarray packages

The standard numarray distribution comes with auxiliary modules that will be quickly outlined here (all these are described in considerably more detail in the numarray manual).

6.6.1 random_array

The `random_array` module is used to generate arrays of random values with a variety of distributions as well as seed-related functions. The following lists the available functions with a couple examples at the end. Functions that return random values all take a `shape` argument for the desired array shape.

`seed(x=0, y=0)`: sets the seed values of the random number generator

`get_seed()`: return the seeds used by the random number generator

`random(shape=[])`: returns Float array of uniformly distributed random numbers between 0. and 1. (not including)

`uniform(min, max, shape=[])`: returns uniformly-distributed Float values between specified min and max (which themselves may be arrays)

`randint(min, max, shape=[])`: like uniform, but returns int values.

`permutation(n)`: returns all n values from 0 to n-1 with order randomly permuted

Other supported distributions are: (Float) `beta`, `chi_square`, `exponential`, `F`, `gamma`, `multivariate_normal`, `normal`, `noncentral_chi_square`, `noncentral_chi_square`, `standard_normal`, and (Int) `binomial`, `negative_binomial`, `multinomial`, `poisson`. In general, the parameters for these distributions may be arrays.

Examples:

```

>>> import numarray.random_array as ra
>>> ra.random((3,3)) # trivial example

```

Example of generating a poisson noise image of a gaussian psf

```

>>> y, x = indices((11,11))
>>> y -= 5 # example of "augmented operator, subtract 5 from y and place result in y
>>> x -= 5
>>> im = 1000*exp(-(x**2+y**2)/2**2) # noiseless gaussian
>>> nim = ra.poisson(im)
>>> print nim

```

6.6.2 fft

This provides the common FFT functions:

`fft(a, n=None, axis=-1)`: Standard 1-d fft (applies fft only 1-dimension of multidimensional arrays)

`inverse_fft(a, n=None, axis=-1)`: Standard inverse 1-d fft

`real_fft(a, n=None, axis=-1)`: Only accepts real inputs and returns half of the symmetric complex transform.

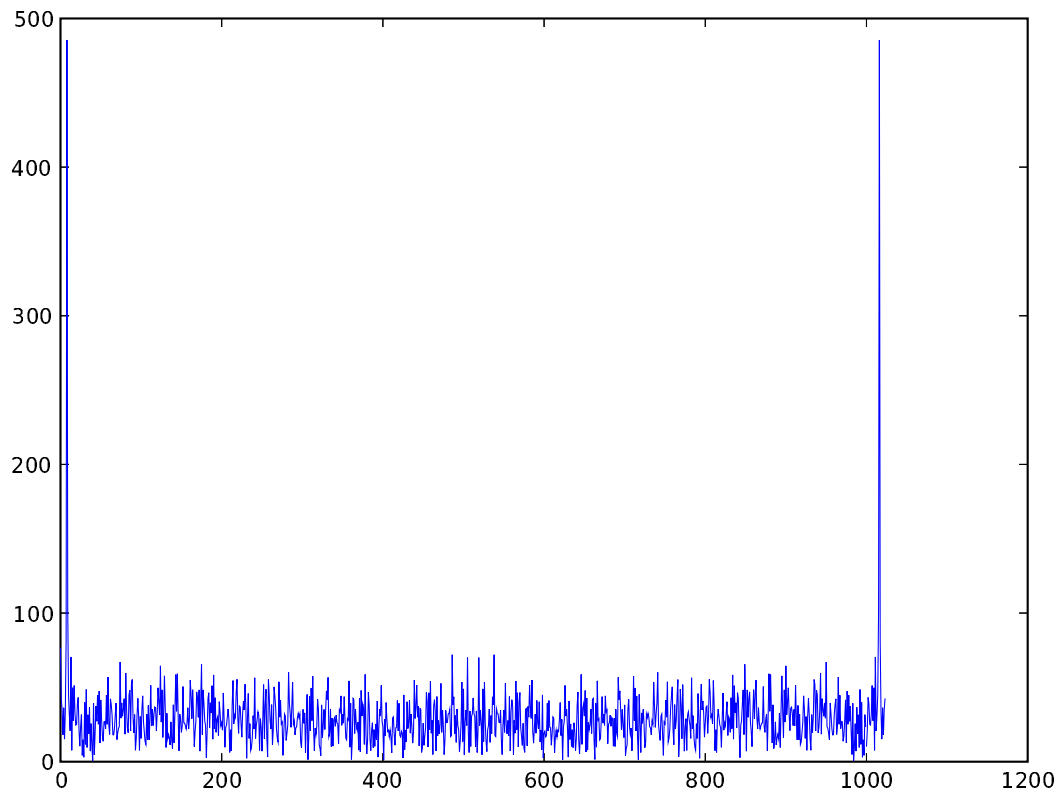
`inverse_real_fft(a, n=None, axis=-1)`: Only accepts real inputs and returns half of the symmetric complex transform.

`fft2d(a, s=None, axes=(-2,-1))`: 2-d fft.

`real_fft2d(a, s=None, axes=(-2,-1))`:

Example:

```
>>> from numpy.fft import fft
>>> x = sin(arange(1024)/20.) + ra.normal(0,1,shape=(1024,))
>>> # sine wave with gaussian noise
>>> from pylab import *
>>> plot(abs(fft(x)))
```



6.6.3 convolve

This includes smoothing functions:

`boxcar(data, boxshape, output=None, mode='nearest', cval=0.)`: standard 1-d and 2-d boxcar smoothing.

`convolve(data, kernel, mode=FULL)`: 1-d convolution function

`convolve2d(data, kernel, output=None, fft=0, mode='nearest', cval=0.)`: 2-d convolution with option to use FFT.

`correlate(data, kernel, mode=FULL)`: 1-d correlation function

`correlate2d(data, kernel, output=None, fft=0, mode='nearest', cval=0.)`: 2-d correlation function with option to use FFT

These functions have various options that determine the size of the resulting array (FULL, PASS, SAME, VALID)

6.6.4 Linear algebra

This includes the common linear algebra operations. The module is `numpy.linalg`. The functions available are (details and examples in `numpy` manual):

`cholesky_decomposition(a)`:

`determinant(a)`

`eigenvalues(a)`

`eigenvectors(a)`

`generalized_inverse(a, rcond=1e-10)`: aka pseudo-inverse or Moore-Penrose inverse

`Heigenvalues(a)`: the real positive eigenvalues of a square, Hermitian positive definite matrix.

`Heigenvectors(a)`: likewise, the eigenvalues and eigenvectors for Hermitian matrices

`inverse(a)`

`linear_least_squares(a, b, rcond=1e-10)`

`solve_linear_equations(a, b)`

`singular_value_decomposition(a, full_matrices=0)`

6.6.5 Masked arrays

The masked array facility allows associating data masks with arrays that will be appropriately propagated when using ufuncs (though many of the add-on modules, e.g., `fft` or `matplotlib`, will not handle the masks implicitly). The use of masks impacts performance (and memory as well), but can be handy if systematic use of masks is required. Note that use of IEEE-754 special values (e.g., Inf, NaNs) can be a faster way of representing bad data values. Use of these IEEE special values will be covered in the next tutorial.

The masked array facility is found in `numpy.ma`. The details of its use are too long to describe here. Only a simple example will be shown.

```

>>> import numpy.ma as ma
>>> x = arange(3) # regular array
>>> y = ma.array([1, 2, 3], mask = [0, 1, 0]) # masked array, 2nd element invalid
>>> x+y
array(data =
      [1 0 5],
      mask =
      [0 1 0],
      fill_value=0)

```

Details of its use can be found in the `numpy.ma` manual.

6.6.6 Multi-dimensional array processing

This module provides a whole host of functions for image processing functions that apply to any number of dimensions including various kinds of filters (linear and non-linear), spatial operations and morphological functions. See the `numpy.ma` manual for details.

6.7 Intermediate `numpy` topics

6.7.1 The Zen of array programming

The trick to making Python data processing efficient for large data sets is the avoidance of loops over small amounts of data (or worse, individual array points). Not that you can't do it. By all means, go ahead. But don't be surprised if the program runs very slowly. Since Python is interpreted, there is a certain overhead to each interpreted Python statement. Efficiency is achieved by avoiding large number of repetitions of Python statements that do relatively little work and instead using operations that operate on the whole array. The following is a simple illustration using the data from `pix.fits`.

```

>>> im = pyfits.getdata('pix.fits')
>>> # Using the Fortran or C approach
>>> sum = 0L
>>> for j in range(512):
...     for i in range(512):
...         sum = sum + im[j, i] # remember that index order!!
print sum
28394234L
>>> Using the array approach
>>> print im.sum()
28394234L

```

The time taken by the first is over 30 times longer than the second.

Avoiding loops requires a different mindset. Some algorithms are relatively easy to code in an array formulation, others are not quite as easy, and some are very difficult. At some point the complexity of avoiding loops is not worth the trouble and consideration should be given to accepting the overhead of loops or writing the algorithm in C. The skill of avoiding loops is a somewhat acquired art and it helps to see lots of examples to learn all the tricks that are involved. Those experienced at using IDL are likely familiar with most of the techniques. Most of the existing array functions are there to facilitate the avoidance of loops. By far, the most useful function in this regard is `where()`. The solutions to more advanced exercises will show examples of some of the techniques.

6.7.2 The power of mask arrays, index arrays, and `where()`

The simplest and usually the most efficient way of conditionally operating on a subset of points is to use mask arrays (i.e., boolean arrays). One can use such mask arrays as an index into an array. For example, to set all points in `im` that are greater than 100 to 100:

```
>>> im[im>100] = 100
```

In a similar vein the following will select all those values greater than 100 and generate a new array of those values:

```
>>> bigvalues = im[im>100]
```

The following illustrates how to use mask arrays to do a crude cosmic ray rejection algorithm:

```
>>> bigdiffmask = abs(im1-im2) > nsigma*sqrt(min(im1, im2))
```

The mask can now be used to set the values of the image with the larger value to the value of the other image.

```
>>> im1mask = bigdiffmask & (im1>im2)
>>> im1[im1mask] = im2[im1mask]
>>> im2mask = bigdiffmask * (im2>im1)
>>> im2[im2mask] = im1[im2mask]
```

Some problems need use of explicit index arrays instead, either because some functions only return index arrays (E.g., `argsort()`) or the indices will be manipulated or processed in some way. As previously mentioned, index arrays can be used as indices as well (hence their name!). The `where()` function is used to turn a mask into index arrays.

Use with one-dimensional arrays is the most straightforward. What isn't quite so obvious is that `where()` does not return an array, but rather a tuple of arrays (the reasons for this are explained a little bit later). Thus to manipulate the array(s) returned by `where`, one needs to index the tuple with the appropriate index. This isn't necessary if the tuple is to be used as an index to another array. To take a very simple example:

```
>>> x = arange(10)
>>> ind = where(x > 5)
>>> x[ind] # this works
array([6, 7, 8, 9])
>>> ind.shape # this doesn't
[...]
AttributeError: 'tuple' object has no attribute 'shape'
>>> ind
(array([6, 7, 8, 9],)
>>> ind[0].shape # this does
(4,)
>>> len(ind) # this is effectively the number of dimensions
1
>>> len(ind[0]) # this is the number of points and is likely what was desired
4
```

Having `where()` return tuples of arrays is more awkward in the 1-d case, however, it is the most natural solution for multidimensional arrays, and there is the side benefit that 1-d and n -d arrays are handled the same way. No doubt that new users will bump into this issue at least a few times.

```
When multidimensional arrays are used, where() returns an index for each dimension as part of the t
>>> indices = where(im > 300)
>>> indices
(array([ 31,  31,  32, ..., 505, 506, 506]),
 array([319, 320, 318, ..., 208, 206, 207]))
>>> im[indices[0], indices[1]]=300 # the explicit approach of using
                                # the component index arrays
>>> im[indices] = 300 # the 'magic' approach, exactly equivalent to the explicit form
```

The examples that follow will illustrate how mask and index arrays may be used to avoid explicitly looping over all elements in an array. You may note that in many cases one could use index or mask arrays. Generally speaking, if mask arrays will do, use them instead. Only when index arrays are much smaller than the original arrays will they be faster than using a mask array.

6.7.3 1-D polynomial interpolation example

We expect interpolation routines to soon become available so the following example should not be necessary to follow to do interpolation. But it remains a good illustration of how it is possible to avoid looping over array elements. Suppose one has a pair of arrays that represent x,y functional pairs. The x array represents x samples for some smooth function and the y array are the values of that function at those x values. We presume that the x values are in increasing order (if they weren't already, use of sort functions could easily reorder the x and y arrays to make that so). Given a third array, xx, of x values for which interpolated values of the function are desired, determine the interpolated values. To keep it reasonably simple, this example will use linear interpolation.

```
>>> x = arange(10.)* 2
>>> y = x**2
>>> import numpy.random_array as ra
>>> xx = ra.uniform(0., 18., shape=(100,)) # 100 random numbers between 0 and 18
>>> xind = searchsorted(x, xx)-1 # indices where x is next value below that in xx
>>> xfraction = (xx - x[xind])/(x[xind+1]-x[xind]) # fractional distance between x points
>>> yy = y[xind] + xfraction*(y[xind+1]-y[xind]) # the interpolated values for xx
```

The same approach can be generalized to higher-order interpolation functions or multiple dimensions.

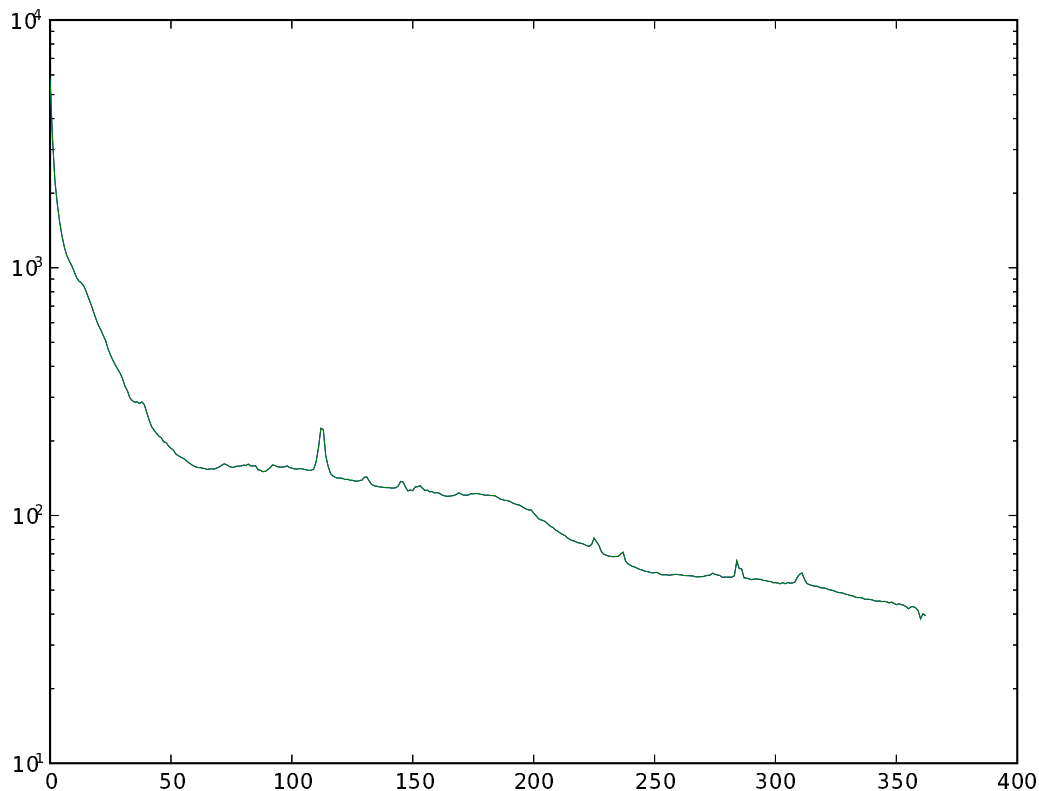
6.7.4 Radial profile example

Suppose one wants to compute the average radial profile for a galaxy (say, M51). The basic problem is to bin image values into radial distance bins and average all the values that fall into the same bin. How can one average all the values at the same radial bin without looping through all radial values? The following shows a way that is faster (but not as fast as one could do in C).

```
>>> y, x = indices((512,512)) # first determine radii of all pixels
>>> r = sqrt((x-257.)**2+(y-258)**2)
>>> ind = argsort(r.flat) # get sorted indices (could use sort
                        # if not needed to arrange image values too)
>>> sr = r.flat[ind] # sorted radii
>>> sim = im.flat[ind] # image values sorted by radii
>>> ri = sr.astype(Int16) # integer part of radii (bin size = 1)
```

Things get trickier here; find numbers in same radius bin by looking for where radius change value and determining distance between changes

```
>>> deltar = ri[1:] - ri[-1] # assume all radii represented
                        # (more work if not)
>>> rind = where(deltar)[0] # location of changed radius
>>> nr = rind[1:] - rind[:-1] # number in radius bin
>>> sim = sim*1. # turn into double to avoid integer overflow
                        # (and single precision rounding)
>>> csim = cumsum(sim) # cumulative sum to figure out sums for each radii bin
>>> tbin = csim[rind[1:]] - csim[rind[:-1]] # sum for image values in radius bins
>>> radialprofile = tbin/nr # the answer
>>> semilog(radialprofile)
```



Well, almost. A careful look at what happens shows that the 0 pixel radius bin is excluded from the final array. It isn't hard to prepend the single value to radial profile.

This is a case where doing it with array operations isn't a great deal simpler than it would be in C. Performance isn't as good largely because of the cost of doing a sort on radius values (not necessary in C). On the other hand, you don't need to learn C, don't need to deal with the hassles of compiling and linking (particularly if you want to share the code), don't need to deal with pointer errors in C, and so forth. So it is often still desirable to solve problems this way.

6.7.5 Random ensemble simulation example

Given a coin, what are the odds of tossing 2 heads in a row before 2 tails given respective probabilities of p and q for heads and tails? This can be computed analytically, but serves as a reasonably simple example of a monte-carlo simulation that can be done with arrays. The general approach to solving this problem is to iterate over every coin toss, but to start with an ensemble of coins. As the iteration continues, some of the coins will hit one or the other termination condition and fewer cases will remain to iterate. For this case, it is necessary to keep the state of the previous coin toss.

```
>>> p = .4
>>> n = 1000000 # initial ensemble size
>>> n2heads = 0; n2tails = 0 # totals for each case
>>> import numpy.random_array as ra
>>> prev = ra.random((n,)) < p # true if heads, false if not
>>> while n>0:
...     next = ra.random((n,)) < p
...     notdone = (next != prev)
```

```

...     n2heads += sum(1.*(prev & next)) # careful about sums!
...     n2tails += sum(1.*logical_not(prev | next))
...     prev = next[notdone]
...     n = len(prev)
...     print n
...
480311
239661
[...]
1
1
0
>>> n2heads, n2tails
(336967.0, 663033.0)

```

This works efficiently so long as many points remain. For the last few iterations where there are few points, the overhead of doing array computations will increase inefficiency. So long as there are relatively few of these, this algorithm should have speeds comparable to those of C. (By the way, the analytically-derived probability is $p^2(1+q)/(1-pq)$.)

6.7.6 thermal diffusion solution example

The following shows how one can apply an iterative solution for Laplace's equation. We construct a 500x500 grid and set a 100x100 pixel boundary condition of 100 at the center and 0 at the edges.

```

>>> grid = zeros((500,500),Float32)
>>> prevgrid = grid.copy()
>>> grid[200:300,200:300] = 100.
>>> done = False
>>> i = 0
>>> while not done:
...     i +=1
...     prevgrid[:,:] = grid # just reuse arrays
...     # new value is average of 4 neighboring values
...     grid[1:-1,1:-1] = 0.25*(
...         grid[:-2,1:-1]
...         + grid[2:, 1:-1]
...         + grid[1:-1,-2]
...         + grid[1:-1,2:])
...     grid[200:300,200:300] = 100.
...     diffmax = abs(grid - prevgrid).max()
...     print i, diffmax
...     if diffmax < 0.1: done = True
1 25.0
2 12.5
[...]
243 0.0996131896973

```

6.7.7 Finding nearest neighbors

Suppose one has a list of coordinates (lets presume simple x,y cartesian) and we want to find the nearest coordinate to each of the coordinates in the list. If the coordinates are provided as x, y arrays of equal length it is possible to use broadcasting to do this quite easily:

```

>>> x = array([1.1, 1.8, 7.3, 3.4])

```

```

>>> y = array([2.3, 9.3, 1.5, 5.7])
>>> deltax = x - reshape(x,(len(x),1) # computes all possible combinations
>>> deltax = y - reshape(y,(len(y),1) # could also use subtract.outer()
>>> dist = sqrt(deltax**2+deltay**2)
>>> dist = dist + identity(len(x))*dist.max() # eliminate self matching
>>> # dist is the matrix of distances from one coordinate to any other
>>> print argmin(dist) # the closest points corresponding to each coordinate
[3 3 3 1]

```

This approach works well so long as there aren't too many points (since it creates arrays that have that number squared entries)

6.7.8 Cosmic ray detection in single image

This example uses a local STScI package and `nd_image` to identify cosmic rays

```

>>> import imagestats
>>> import pyfits
>>> from numarray import nd_image as nd
>>> chip = pyfits.getdata('acs.fits','sci',1)
>>> # Perform morphological closing and opening to filter image of any
>>> # source smaller than (5,5).
>>> # A size of (3,3) still left some residuals for ACS data
>>> closechip = nd.grey_closing(chip,size=(5,5))
>>> occhip = nd.grey_opening(closechip,size=(5,5))
>>> # compute sigma of background
>>> sigma = imagestats.ImageStats(occhip,nclip=3).stddev
>>> # create mask using difference between original input and filtered image.
>>> mask = (chip - occhip) > sigma

```

6.7.9 Source extraction

Extract magnitudes and positions of sources in image (similar to `daofind`)

```

>>> import pyfits
>>> import imagestats
>>> import numarray.nd_image as nd
>>> from numarray import *
>>> sci, hdr = pyfits.getdata('acs2.fits','sci',1,header=True)
>>>
>>> # compute background value, clipping out bright sources
>>> scistats = imagestats.ImageStats(sci,nclip=3)
>>>
>>> # create a binary image flagging all sources in image
>>> # with values greater than the mean of the background
>>> # For crowded fields, extra care/work will need to be
>>> # done to separate blended sources.
>>> sci_clip = sci >= scistats.mean
>>>
>>> label each source with its own ID index
>>> sci_labels,sci_num = nd.label(sci_clip)
>>>
>>> # Compute a sum for each object labeled and return as a list
>>> counts = nd.sum(sci,sci_labels,range(sci_num))
>>>

```

```

>>> # Get position of object using its center of mass
>>> pos = nd.center_of_mass(sci,sci_labels,range(sci_num))
>>>
>>> # Retrieve photometric keywords from image header
>>> photflam = hdr['photflam']
>>> photzpt = hdr['photzpt']
>>> # Convert counts to magnitudes for each object
>>> mag = photzpt - 2.5*log10(array(counts)*photflam)

```

6.7.10 Other issues regarding efficiency and performance

Even if a problem can be cast into a purely array manipulation form, one may encounter other barriers to performance. The most likely is that you will run into memory limitations. If the algorithm involves computations on large arrays, and many temporary or ancillary arrays are needed as part of the computation, then one can easily run out of available memory, or if not virtual memory, real memory, leading to unacceptable system paging. If the algorithm essentially is “ufunc”-like, that is computations are performed independently on pixels, then the array can be segmented and iterated over the segments. For two dimensional arrays in IDL this typically meant iterating over rows in images. That can be done in numarray as well, but there are other approaches that can be used. Even when algorithms are not strictly independent of neighboring pixels, so long as the algorithm is reasonably local, then one can iterate over segments that overlap to the necessary degree. Many problems fall in this category. A solution of this sort will be illustrated in tutorial 4.

Some algorithms result in many array creations and destructions. Even though in theory no memory is being consumed over the long run, it is possible to eventually fragment the available memory as to prevent allocating arrays of sufficient size (this is a problem with IDL and many other systems as well). In such cases it makes sense to reuse existing arrays (if sizes don't change) in iterations. This can be done by assigning to an existing array through slice assignment. For example:

```

>>> im = 2*im # replaces the array im used to refer to
           # with a new array
>>> im[:,:] = 2*im # writes back into the array im
>>> im *= 2 # also reuses im array
           # (but be careful of type downcasts!)
>>> multiply(im, 1, im) # use of optional output array

```

Exercises

1. Read each of the image extensions in the FITS file `tut3f1.fits` using the object-oriented interface, assemble an image cube from the 2-d images in each of the extensions (they are all the same size), and without loops, generate a 2-d result that contains the maximum value at that pixel for the whole stack of images. Display the result. The answer will be obvious if you did it correctly.
2. Add the keyword/value pair `IMNUM= <extension number>` to each of the headers `tut3f1.fits` right after the `TARGET` keyword. While you are at it, add a history keyword. Verify the headers have been updated with your favorite FITS header viewing tool (i.e, not PyFITS)
3. Using the random number module, generate a million random points with x,y coordinates between 0,1. What fraction of these points lie within a radius of 1 from the origin? How close is this to the expected result?
4. The irregularly sampled Discrete Fourier Transform (DFT) is given by

$$F(\omega) = \sum_{n=0}^{N-1} f_n e^{-i\omega n \Delta t}$$

Given an array of time samples f_n , desired frequencies to sample ω_m (not necessarily uniformly spaced), and $\Delta t = 1$ compute the DFT at those frequencies without using any loops.

Acknowledgements

Thanks to Phil Hodge, JC Hsu, Todd Miller, Warren Hack, 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://stdas.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://stdas.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 (IDLDE)
- 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
@<filename>	ipython: run execfile('fileame')
exit	ipython: run control-D (MS windows: control-Z)
up-arrow (command recall)	ipython: Quit or Exit up-arrow
Operators	
<, > (clipping)	Currently no operator equivalent. The functional equivalent is <code>choose(a<100, (a, 100.))</code>
a < 100.	
a > 100.	
MOD	%
# (matrix multiply)	<code>multiply.outer()</code> is equivalent for some applications, <code>matrixmultiply()</code> 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!
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	
i:j	Note: order of indices is opposite! a(i,j) (IDL) equivalent to a[j,i] (Python)
i:*	i:j (j element not part of slice)
*:i	i:
a(i,*)	:i
a(i:j:s)	a[:,i] 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)
arr(indexarr) = valuearr
```

```
newarr = arr[indexarr]
arr[indexarr] = valuearr
```

Array Creation

```
fltarr()
dblarr()
complexarr()
intarr()
longarr()
bytarr()
make_arr()
strarr()
findgen()
dindgen()
indgen()
lindgen()
sindgen()
replicate()
```

```
array(seq, [type]) to create from existing sequence
zeros(shape, [type]) to create 0 filled array
ones(shape, [type]) to create 1 filled array
chararray for fixed length strings
```

```
arange(size, [type])
arange(start, end, [type])
arange(start, end, step, [type])
no equivalent for strings

repeat (more general)
```

Array Conversions

```
byte(arrayvar)
fix(arrayvar)
long(arrayvar)
float(arrayvar)
double(arrayvar)
complex(arrayvar)
```

```
arrayvar.astype(<type>)
```

Array Manipulation

```
reform()

sort()
arr(sort(arr))
max(arrayvar)
min(arrayvar)
where()
arr(where(condition))
transpose()
[a,b] (array concatenation)
```

```
reshape() [also arrayvar.flat() and ravel(), or
changing the shape attribute]
argsort() [i.e., indices needed to sort]
sort(arr) [i.e., sorted array]
arrayvar.max()
arrayvar.min()
where()
arr[where(condition)]
transpose()
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: matplotlib for IDL Users

This is not a complete discussion of the power of matplotlib (the OO machinery) or pylab (the functional interface). This document only provides a translation from common IDL plotting functionality, and mentions a few of the additional capabilities provided by pylab. For a more complete discussion, you will eventually want to see the tutorial and the documentation for the pylab interface:

```
http://matplotlib.sourceforge.net/tutorial.html
http://matplotlib.sourceforge.net/matplotlib.pylab.html
```

Setting up and customizing your Python environment:

Whether you use PyRAF, ipython, or the default python interpreter, there are ways to automatically import your favorite modules at startup using a configuration file. See the documentation for those packages for details. The examples in this document will explicitly import all packages used.

Setting up and customizing matplotlib:

You will want to modify your `.matplotlibrc` plot to use the correct backend for plotting, and to set some default behaviors. (The STScI versions of the default `.matplotlibrc` have already been modified to incorporate many of these changes.) If there is a `.matplotlibrc` file in your current directory, it will be used instead of the file in your home directory. This permits setting up different default environments for different purposes. You may also want to change the default behavior for image display to avoid interpolating, and set the image so that pixel (0,0) appears in the lower left of the window.

```
image.interpolation : nearest # see help(imshow) for options
image.origin       : lower   # lower | upper
```

I also wanted to change the default colors and symbols.

```
lines.marker       : None    # like !psym=0; plots are line plots
                    by default
lines.color        : k       # black
lines.markerfacecolor : k    # black
lines.markeredgecolor : k    # black
text.color         : k       # black
```

Symbols will be solid colored, not hollow, unless you change `markerfacecolor` to match the color of the background.

About overplotting behavior:

The default behavior for matplotlib is that every plot is a "smart" overplot, stretching the axes if necessary; and one must explicitly clear the axes, or clear the figure, to start a new plot. It's possible to change this behavior in the setup file, to get new plots by default

```
axes.hold         : False    # whether to clear the axes by
                        default on
```

and override it by specifying the "hold" keyword in the plotting call. However I DO NOT recommend this; it takes a bit of getting used to, but I found it easier to completely change paradigms to "build up the plot piece by piece" than it is to remember which commands will decorate an existing plot and which won't.

Some philosophical differences:

Matplotlib tends towards atomic commands to control each plot element independently, rather than keywords on a plot command. This takes getting used to, but adds versatility. Thus, it may take more commands to accomplish something in matplotlib than it does in IDL, but the advantage is that these commands can be done in any order, or repeatedly tweaked until it's just right. This is particularly apparent when making hardcopy plots: the mode of operation is to tweak the plot until it looks the way you want, then press a button to save it to disk. Since the pylab interface is essentially a set of convenience functions layered on top of the OO machinery, sometimes by different developers, there is occasionally a little bit of inconsistency in how things work between different functions. For instance the errorbar function violates the principle in the previous paragraph: it's not an atomic function that can be added to an existing plot; and the scatter function is more restrictive in its argument set than the plot function, but offers a couple of different keywords that add power.

Some syntactic differences:

You can NOT abbreviate a keyword argument in matplotlib. Some keywords have had some shorthand versions programmed in as alternates - for instance `lw=2` instead of `linewidth=2` - but you need to know the shorthand. Python uses indentation for loop control; leading spaces will cause an error if you are at the interactive command line. To issue 2 commands on the same line, use a semicolon instead of an ampersand.

The Rosetta Stone:

	IDL	Matplotlib
Setup	(none needed if your paths are correct)	<pre>from numpy import * from pylab import * import pyfits</pre> (Or, none needed if you set up your favorite modules in the configuration file for the Python environment you're using.)
Some data sets:	<pre>foo = indgen(20) bar = foo*2</pre>	<pre>foo = arange(20) bar = foo*2</pre>
Basic plotting:	<code>plot, foo</code>	<code>plot(foo)</code>
	<code>plot, foo, bar</code>	<code>plot(foo, bar)</code>
	<code>plot, foo, bar, line=1, thick=2</code>	<pre>plot(foo, bar, '-', lw=2) ***or spell out linewidth=2</pre>
	<code>plot, foo, bar, ticklen=1</code>	<pre>plot(foo, bar) grid()</pre>
	<code>plot, foo, bar, psym=1</code>	<pre>plot(foo, bar, 'x')</pre> OR <pre>scatter(foo, bar)</pre>
	<code>plot, foo, bar, psym=-1, symsize=3</code>	<pre>plot(foo, bar, '-x', ms=5) ***or spell out markersize=5</pre>
	<code>plot, foo, bar, xran=[2,10], yran=[2,10]</code>	<pre>plot(foo, bar) xlim(2,10) ylim(2,10)</pre>
	<code>err=indgen(20)/10.</code>	<code>err = arange(20)/10.</code>

	<pre>plot, foo, bar, psym=2 errplot, foo, bar-err,bar+err</pre>	<pre>errorbar(foo, bar, err, fmt='o') errorbar(foo, bar, err, 2*err, 'o') (error bars in x and y)</pre>
Overplotting with default behaviour changed (Not recommended or used in any other examples)	<pre>plot, foo, bar oplot, foo, bar2</pre>	<pre>plot(foo, bar) plot(foo, bar2, hold=True)</pre>
Text, titles and legends:	<pre>xyouts, 2, 25, 'hello'</pre>	<pre>text(2, 25, 'hello')</pre>
	<pre>plot, foo, bar, title='Demo', xtitle='foo', ytitle='bar'</pre>	<pre>plot(foo, bar) title('Demo') xlabel('foo') ylabel('bar')</pre>
	<pre>plot, foo, bar*2, psym=1 oplot, foo, bar, psym=2</pre>	<pre>plot(foo, bar*2, 'x', label='double') plot(foo, bar, 'o', label='single')</pre>
	<pre>legend,['twicebar','bar'], psym=[1,2],/upper,/right</pre>	<pre>legend(loc='upper right') ***legend will default to upper right if no location is specified. Note the *space* in the location specification string</pre>
	<pre>plot, foo, bar, title=sysptime()</pre>	<pre>import time plot(foo, bar) label(time.asctime())</pre>
Window Manipulation	<pre>erase</pre>	<pre>clf() OR cla()</pre>
	<pre>window, 2</pre>	<pre>figure(2)</pre>
	<pre>wset, 1</pre>	<pre>figure(1)</pre>
	<pre>wdelete, 2</pre>	<pre>close(2)</pre>
	<pre>wshow</pre>	<pre>[no equivalent]</pre>
Log plotting	<pre>plot_oo, foo, bar</pre>	<pre>loglog(foo+1, bar+1)</pre>
	<pre>plot_io, foo, bar</pre>	<pre>semilogy (foo, bar)</pre>
	<pre>plot_oi, foo, bar</pre>	<pre>semilogx (foo, bar)</pre>
		<pre>***Warning: numbers that are invalid for logarithms (<=) will not be plotted, but will silently fail; no warning message is generated. ***Warning: you can't alternate between linear plots containing zero or negative points, and log plots of valid data, without clearing the figure first - it will generate an exception.</pre>

Postscript output	<pre>set_plot, 'ps' device, file='myplot.ps',/land plot, foo, bar device,/close</pre>	<pre>plot(foo, bar) savefig('myplot.ps', orientation='landscape')</pre>
Viewing image data:	<pre>im = mrdfits('myfile.fits', 1) tv, im loadct, 5 loadct, 0 contour, im xloadct</pre>	<pre>f = pyfits.open('myfile.fits') im = f[1].data imshow(im) jet() gray() contour(im) [no equivalent]</pre>
Histograms	<pre>mu=100 & sigma=15 & x=fltarr(10000) seed=123132 for i = 0, 9999 do x[i] = mu + sigma*randomn(seed) plothist, x</pre>	<pre>mu, sigma = 10, 15 x = mu + sigma*randn(10000) n, bins, patches = hist(x, 50)</pre>
	<pre>*** Note that histograms are specified a bit differently and also look a bit different: bar chart instead of skyline style</pre>	
Multiple plots on a page	<pre>!p.multi=[4,2,2] plot, foo, bar plot, bar, foo plot, foo, 20*foo plot, foo, bar*foo</pre>	<pre>subplot(221) ; plot(foo, bar) subplot(222) ; plot(bar, foo) subplot(223) ; plot(foo, 20*foo) subplot(224) ; plot(foo, bar*foo)</pre>
Erasing and redrawing a subplot:	[no equivalent]	<pre>subplot(222) cla() scatter(bar, foo)</pre>
plotting x, y, color and size:	[no obvious equivalent]	<pre>scatter(foo, bar, c=foo+bar, s=10*foo)</pre>
Adding a colorbar	[no obvious equivalent]	<pre>colorbar()</pre>

Some comparisons:

Some additional functionality:

- modify axes without replotting
- add labels, generate hardcopy, without replotting
- colored lines and points, many more point styles
- smarter legends
- \TeX -style math symbols supported
- interactive pan/zoom

Some functionality that is not yet conveniently wrapped:

(These are items that are available through the OO machinery, but are not yet wrapped into convenient functions for the interactive user. We welcome feedback on which of these would be most important or useful!)

- subtitles
- loadct - to dynamically modify or flick through colortables
- tvrdc,x,y - to read the xy position of the cursor (will soon be available)
- histogram specification by bin interval rather than number of bins

Some still-missing functionality:

(These are items for which the machinery has not yet been developed.)

- surface plots
- save - to save everything in an environment (will soon be available)
- journal - to capture commands & responses. (ipython can capture commands.)

Symbols, line styles, and colors:

!psym equivalences:

- 0** line -
- 1** plus +
- 2** asterisk unsupported; overplotting + with x is close
- 3** dot .
- 4** diamond d
- 5** triangle ^
- 6** square s
- 7** cross x
- 10** histogram ls='steps'

!linestyle equivalences:

- 1** solid -
- 2** dotted :
- 3** dashed -
- 4** dash-dot -.

The following line styles are supported:

- : solid line
- : dashed line
- . : dash-dot line

: : dotted line
. : points
, : pixels
o : circle symbols
^ : triangle up symbols
v : triangle down symbols
< : triangle left symbols
> : triangle right symbols
s : square symbols
+ : plus symbols
x : cross symbols
D : diamond symbols
d : thin diamond symbols
1 : tripod down symbols
2 : tripod up symbols
3 : tripod left symbols
4 : tripod right symbols
h : hexagon symbols
H : rotated hexagon symbols
p : pentagon symbols
| : vertical line symbols
_ : horizontal line symbols
steps : use gnuplot style 'steps' # kwarg only

The following color strings are supported

b : blue
g : green
r : red
c : cyan
m : magenta
y : yellow
k : black
w : white

Matplotlib also accepts rgb and colorname specifications (eg hex rgb, or "white").

Approximate color table equivalences:

loadct,0 gray()

loadct,1 almost bone()

loadct,3 almost copper()

loadct,13 almost jet()

Color tables for images:

autumn

bone

cool

copper

flag

gray

hot

hsv

jet

pink

prism

spring

summer

winter