Detection of Light

A Quick Introduction to Python

Some useful links...

- The Python tutorial
  - http://docs.python.org/tutorial/
  - This tutorial introduces the reader informally to the basic concepts and features of the Python language and system.
- STScI Python Data Analysis Tutorial
  - http://www.scipy.org/Additional_Documentation/Astronomy_Tutorial?action=show&redirect=wikis%2Ftopical+software%2FTutorial
  - Written by and for astronomers, this thorough introduction will take you through several examples of manipulating and plotting astronomical data, including handling FITS files. The web site includes a bundle of example data sets as well as the tutorial document.

Some useful links...

- STScI Pyfits Tutorial
  - http://packages.python.org/pyfits/users_guide/users_tutorial.html
  - A quick introduction of using PyFITS. The goal is to demonstrate PyFITS's basic features without getting into too much detail. If you are a first time user or an occasional PyFITS user, using only the most basic functionality, this is where you should start.
- AstroBetter's IDL to Python cheat sheet
  - http://www.astrobetter.com/wiki/tiki-index.php?page=Python+Switchers+Guide
  - This page lists Pythonic versions of some of the more common commands from the Goddard IDL astronomy library, plus some useful more general translations for e.g. plotting and IO commands.

Standard libraries

- We’re going to mostly read our data from FITS files, using a module called **pyfits**
  - http://www.stsci.edu/resources/software_hardware/pyfits
- We’ll crunch up the data using a module called **numpy**
  - http://numpy.scipy.org
- For graphical output we’ll use the module **matplotlib**
  - http://matplotlib.org/

Python

- Python uses ‘block indenting’:
  - if you start any block, like an if statement, or a for loop, you have to indent within the block. The relaxation of this indenting is the way python recognizes the end of the block.
- You don’t have to run python as a script, you can also use it interactively
  - Type ipython at your normal unix prompt and you’ll start an interactive python session. ipython –pylab will load the matplotlib plotting library too
  - You can enter your python statements then line by line. Type ‘control-d’ to get out when you’re finished.

Ipython

- Linux commands

```
> pwd
> ls
```

- Run scripts

```
> %run test.py
```

  - The script variables remain in the environment. Useful for debugging.

```
> print data
```

- Tab completion

Python variables

- Scalar types:
  - Real
  - Integer
  - String
  - Boolean
  - Object
- BUT the python philosophy is to ignore this distinction as far as possible. Variables are not ‘declared’ as with other languages. Python sets (or resets, if necessary) the type of a variable to fit whatever data you try to load into it.

Python variables

- Ways to arrange several objects:
  - A single 'scalar': 3.14
  - A list: ['mystring', 3.14, ...]
  - A tuple: ('mystring', 3.14, ...)
  - A dictionary:elements accessed by key, data types can still be different {'string': 'mystring', 'pi': 3.14, ...}
  - A numpy array: np.array((1., 2., 3., 4.))
- Type

```
> x=[1.2, 1, 'string']
> type(x)
list
```

Python control structures

- If, then, else

```
if <test1>:
   # do some stuff
elif <test2>: # you can have 0 or more of these
   # do some other stuff
else: # you can have 0 or 1 of these
   # third lot of stuff
```

Python control structures

- While

```
while <test>:
    # do loop stuff
    break # optional – dumps out and avoids the ‘else’.
    continue # like ‘goto while’.
else: # optional - processing after normal loop exit.
    # stuff to do after normal loop exit
```

Python control structures

- For

```
for <item> in <list>: # etc


for i in [1,2,3,4,5]:
    print i
```

Traps

- Python numbering starts at zero!
- 'change in place' of mutable objects

```
> aa=[1,2,3]
> bb=aa
> aa[0]=999
> print bb
 - [999, 2, 3]
```

Traps

- Automatic casting of variables:

```
In [1]: x=1
In [2]: y=2
In [3]: x/y
Out[3]: 0

In [4]: x=1.0
In [5]: x/y
Out[5]: 0.5
```

Numpy

- arrays

```
> x = np.array([1,2,3,4,5,6,7])
> type(x)
numpy.ndarray
> np.shape(x)
(7,)
```

- More dimensions:

```
> x=np.array([[1,2,3,4,5,6,7],[1,2,3,4,5,6,7]])
> m,n = np.shape(x)
> print m, n
2, 7
```

Numpy

- Zeros, ones

```
> x=np.zeros(10)
> x
array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
0.])
> np.ones((10,2))
array([[ 1.,  1.],
       [ 1.,  1.],
       ...
       [ 1.,  1.],
       [ 1.,  1.]])
> x=np.ones((10,2,3,4))
> np.shape(x)
(10, 2, 3, 4)
```

Numpy

- Indexing arrays

```
> x=np.ones((5,3)))
> x[0,0]
- 1.0
- x[0,0]=2
- print x
- x[0,1]=3.
```

Numpy

- arange

```
In [44]: np.arange(1,6,0.5)

Out[44]: array([ 1. ,  1.5,  2. ,
2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
5.5])


In [45]: np.arange(1,6.01,0.5)

Out[45]: array([ 1. ,  1.5,  2. ,
2.5,  3. ,  3.5,  4. ,  4.5,  5. ,
5.5,  6. ])
```

Numpy

- Back to array indexing:

```
> print x
array([[ 2.,  3.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
> m,n=np.shape(x)
> for i in np.arange(m):
    for j in np.arange(n):
        print i,j, x[i,j]
```

Numpy

- Back to array indexing:

```
In [63]: for i2 in np.arange(m):
   ....:          for j2 in np.arange(n):
   ....:                  x[i2,j2] = i2*j2
   ....:

In [64]: x
Out[64]:
array([[ 0.,  0.,  0.],
       [ 0.,  1.,  2.],
       [ 0.,  2.,  4.],
       [ 0.,  3.,  6.],
       [ 0.,  4.,  8.]])
```

Numpy

- Back to array indexing:

```
In [67]: x=np.ones((10,10))

In [68]: m,n=np.shape(x)

In [69]: for i2 in np.arange(m):
    ....:         for j2 innp.arange(n):
    ....:                 x[i2,j2] = i2*j2
```

Numpy

- Array slicing

```
> x[:,2]
> y = x[:,0:4]
```

  - Beware! This doesn't include 4:

```
> range(0,4,1)
```

  - Back to the example

```
> x[:,0:4]*2.3
> x
> x[:,0:4]=x[:,0:4]*2.3
> x[:,0:4]=x[:,0:4]/2.3
```

Numpy

- Reshape

```
> t=np.arange(0,35)
> y = t.reshape((5,7))
```

- average

```
> np.average(y)
> np.average(y, axis=0)  # shape is (7,)
> np.average(y, axis=1)  # shape is (5,)
```

  - axis

- array math

```
> t+3
```

  - `z=np.arange(5)`
  - `y.shape, z.shape`
  - `y-z`

Numpy

```
> y
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, 24, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34]])
> avgy1 = np.average(y,axis=0)
> avgy2 = np.average(y,axis=1)
> y-avgy1  -> works – subtract column average
> y-avgy2  -> ValueError
```

- Trickery:

```
> (y.transpose()-avgy2).transpose()
> y - avgy2[:,np.newaxis]
```

Pyfits

- import pyfits as pf
- pf.<tab> in ipython will get you a list of functions
- Read the data

```
> data = pf.getdata('image1_2014.fits')
> type(data)
```

- Now we can manipulate the numpy array

```
> data.shape
> data
> data[118:139,128:139]
> data[118:139,118:139]
> data[139,139]  -> IndexError!
> data[20:30,15:18]=data[20:30,15:18]*0.
```

Pyfits

- Reading a header

```
> head =
pf.getheader( 'image1_2014.fits' )
```

- Writing fits files

```
> pf.writeto('image1_2014_out2.fits',
data, header=head, clobber=True)
>
```

Matplotlib

- import matplotlib.pyplot as pl

```
> x = pl.arange( 0, 2.*pl.pi, pl.pi/1000 )
> pl.pi
> y = 5. + 0.5*x*pl.sin( x )
> x2 = pl.arange( 0, 2.*pl.pi, pl.pi/20 )
> y2= 5. + 0.5*x2*pl.sin( x2 )
> pl.plot( x, y, 'k', label='line' )
> pl.show()
> pl.plot( x, y, 'k', label='line' )
> pl.plot( x2, y2, 'go', label='points' )
> pl.legend()
> pl.xlabel( '$x$' )  # LaTeX!
> pl.ylabel( '$f(x)$' )
> pl.title( 'Some function of x' )
> pl.savefig( 'dummyplot.png' )
> pl.show()
```

Matplotlib

- imshow

```
> pl.imshow(data)
```

```
> pl.imshow(data,
interpolation='nearest',
origin='lower', cmap='hsv')
```