Python Crib Sheet

Paddy Alton, CEA, Durham University

October 13, 2015

1 Introduction

This is a list of useful Python commands, written over a month after I transferred from using IDL to using Python (having used IDL for over two years prior to that).

Most of the things on the list are things I found helpful enough to bother writing down (since you can always just google it with Python); often they are things which are less trivial to do in Python than in IDL, or which offer big improvements over the IDL way of doing things.

2 Basics

2.1 modules

All Python programmes will start by you importing a bunch of modules, which contain useful functions. The command is

```
import module
```

or

```
import module as mod
```

These functions are then called by

```
module.function(arguments)
```

or

```
mod.function(arguments)
```

You can also do

```
from module import function
function(arguments)
```

2.2 miscellania

Run Python scripts, such as scriptname.py, using

```
>>> run scriptname
```

Comments are marked with # signs, e.g.

```
<code> # comment
```

Also, always use ipython. Just type 'ipython' if it's installed on your machine to start it up, and 'exit' to shut it down. Tab complete works, which is v. useful!

To find out what a variable IS, you can just type it into ipython. For example,

```
>>> array       # returns the values of array
print array     # if in a script (same as ‘PRINT,array’ in IDL)
>>> array.size  # is the equivalent of N_ELEMENTS(array) in IDL
>>> len(array)  # is more or less a synonym of this
>>> array.shape # returns the shape of the array, e.g. (6,4,2)
```

If you're debugging code, you can do a STOP command equivalent using

```
raw_input(‘message that will be printed when you encounter this line’)
```

... and press Enter to continue, or ctrl+C to break out.

Finally, basic operations are what you'd expect, except that x^2 etc. is called by x**2, and to get a remainder (the IDL command is a MOD b) you do a%b. You can do a truncated division (I.E. get the bit which ISN'T the remainder) by using a//b.

2.3 numpy stuff - arrays

‘numpy’ is one of the most useful modules. You'll import it at the start of pretty much any script you write:

```
import numpy as np
```

Here are some things this allows you to do:

```
[<values>] # this is a list. It might not behave the way you want
           # an array to.

arr = np.array( [<values>], dtype=float ) # makes an array of floats

    = np.zeros( (3,2), dtype = float) # makes a 3x2 array of floats

    = np.arange(N) # makes the array [1,2,...,N-1]
```

Now,

```
np.sum(arr) # returns the sum of the elements of ‘arr’

np.sum(arr, axis=M) # returns the sum of the elements of ‘arr’
                    # along the M-th axis

np.mean(arr) # self explanatory
np.median(arr, axis=M) # likewise

np.nanmean() # ignores NaNs
np.nanmedian() # likewise
```

Best of all, this is how you do structures in Python:

```
structure = np.zeros( m, dtype = [ (‘name_1’,str),
                                   (‘name_2’,float),
                                   (‘name_3’,float,(6,4)) ] )
```

Then ‘structure’ has m rows, each containing 1 string, 1 float, and 1 (6×4) array of floats. These are called by

```
array_of_floats = structure[‘name_2’]
```

You can find the tag names using structure.dtype.names, which returns the list

```
[‘name_1’,‘name_2’,‘name_3’]
```

This means you can then do something like:

```
structure[structure.dtype.names[index]]
```

which returns the appropriate output for

```
structure['name_1']
structure['name_2']
structure['name_3']
```

for index values of 0,1, and 2 respectively.

2.4 numpy stuff - other things

Some other useful features of numpy are as follows:

```
np.max(arr)    # max value of an array
np.argmax(arr) # index corresponding to that value
np.min(arr)    # min value of an array

np.sort(arr)          # returns the sorted array
np.argsort(arr)       # returns the indices instead of the values
np.where(<condition>) # equivalent of WHERE() in IDL, but see below...
```

Not forgetting some things you might expect, such as

```
np.exp(x) # returns exp(x)
np.log(x) # returns ln(x)
np.log10(x) # returns log_10(x)

np.pi # returns 3.1415926535897931
```

2.5 array handling (general)

Beyond numpy, Python has a whole bunch of built in stuff that makes dealing with arrays nice and easy. There are some subtleties, and there are difference to IDL.

```
array[0]  # returns first element of array
array[-1] # returns final element of array

array[m:n] # returns the (m+1)-th element, up to the n-th element
           # (so it excludes array[n])

array[0:-1] # returns first to PENULTIMATE element of array
array[0:]   # returns first to last element of array
array[:]    # returns all elements of array

array[9:14:2] # this returns elements 10 to 14, in steps of two.
```

Now for the best part - the bit which more or less makes np.where() obsolete:

```
array[<condition>] # returns elements of array that satisfy the
                   # condition. (N.B. it returns their VALUES,
                   # rather than the INDICES like np.where)

array[ array > 10 ] # returns elements of array which are bigger
                    # than 10.

array[ (array>10)&(array<20) ] # an example of using '&'
                               # (logical AND)

array[ (array<10)|(array>20) ] # an example of using '|'
                               # (logical OR)
```

3 Loops

Obviously at some point you're going to want to do a FOR loop. Python is quite flexible about this, despite everyone telling you that you should never use for loops...

```
# this loop runs over all the values contained in 'array'

for i in array:

    < indented code >

<un-indented code> # closes the FOR loop. Don't forget this!


# the next for loop is more traditional...

for i in range(N):

    < indented code >

# ... it runs from i=0 to i=(N-1).
# N could be something like len(array).
```

It doesn't have to be laid out exactly that way all the time. You can do some really clever stuff like this:

```
[i for i in list if i.endswith('<suffix>')]
```

... to break that down, let's say 'list' is a list of strings. The command runs through the elements of list one by one, naming them 'i', then, if i ends with 'suffix', which might be something like '.fits', i is returned.

```
list = ['thing.fits','thing2.dat','thing3.py','thing4.fits']

new_list = [ i for i in list if i.endswith('.fits') ]

print new_list # returns ['thing.fits','thing4.fits']
```

One can imagine different conditions, but I thought .endswith might be useful to include.

4 File Handling

That last example would actually be a last resort for handling files! There are plenty of things which are already in place which make life convenient.

The first question I asked when I switched to Python, more or less, was "how do I do 'SPAWN,' ???". Here is the answer:

```
import subprocess
subprocess.call("command you want to send",shell=True)

# or

output = subprocess.call("command you want to send",shell=True)
```

Note that you have to unlock the shell, for security reasons (we almost always know exactly what files etc. our commands will act on, but one can envisage a situation where code written this way might act on files with names containing malicious commands - see http://en.wikipedia.org/wiki/Code_injection#Shell_injection for details. Anyway, the point is that it makes sense that you have to do this, honest).

Having said that, if you've been using SPAWN in IDL to do things like 'ls directory', that won't work the way you want it to. Fortunately, someone's always written something to do the thing you want...

```
# return a file list

import os
files = os.listdir('/obs/r2/username_that_makes_no_sense/stuff/')

# ls

import glob
files = glob.glob('~/raw_data/*.fits') # what you'd expect.
```

The os module also does some other useful things, like

```
os.path.isfile('filename.ext') # returns True if filename.ext exists

os.remove('filename.ext') # executes an 'rm' command.
```

Actually, if you REALLY want to run commands in the terminal and access the output, that is possible, though fiddly. Watch carefully:

```
receiver = subprocess.Popen("grep 'searchterm' textfile.txt",
                            stdout=subprocess.PIPE, shell=True)

output = receiver.communicate() # output from the grep is piped to
                                # 'receiver', now we extract it.
```

...and you can do this with other shell commands than 'grep' of course. Though, there is usually a better way!

Finally, you might end up trying to chop bits out of filenames, particularly if you are used to IDL's STRSPLIT. Here's the python way:

```
string = 'some_long_filename_with_underscores.py'

output = string.split("_")

# returns ['some', 'long', 'filename', 'with', 'underscores.py']
```

4.1 FITS files

Most observers will use FITS files at some point, and it turns out that there's a module for that too.

```
import astropy.io.fits as fits

data_structure = fits.getdata('filename.fits')   # read in data
data_structure = fits.getdata('filename.fits',2) # read in data from
                                                 # extension 2

# different ways to get the header from the file:
data_structure, my_header = fits.getdata('filename.fits',header=True)
my_header = fits.getheader('filename.fits')

# writing output ('clobber=True' means overwrite existing file)
fits.writeto('output.fits',output,header=my_header,clobber=True)

fits.append('output.fits',output) # append a new extension rather
                                  # than overwriting file
```

All fairly self-explanatory. Arrays are written out as images, structures are written out as tables. Headers are treated as special objects by python, which allow you to read the value of a particular keyword:

```
print my_header['KEYWORD'] # print the value assigned to KEYWORD
```

5 Functions

Defining functions is quite useful, but what's even better is putting all the small functions you use a lot in one file and calling that a module. So first, here's how to do functions (basic ones - there's a lot of advanced stuff you can do).

```
def function(argument_1, argument_2, ... , argument_n):

    < indented code which does whatever you want the function to do >

    return output_1, output_2, ... , output_m
```

Straightforwardly, you can define functions in this way at the bottom of your main programme and call them in the way you'd expect:

```
output_1, output_2 = function(argument_1, argument_2, ... , argument_n)
```

... or you could put it in a file called 'function.py' and at the start of your programme put in the statement

```
from function import *

# or

from function import function
```

... but this creates problems, namely that Python is a bit lazy and only compiles functions once per ipython session, and won't update the function unless told to. You can tell it to do that like this:

```
import imp
imp.reload(function)
```

However, by far the best way to do things is to make a file called something like 'mymodule.py', put ALL the functions you ever use in there, then do

```
import imp
import mymodule as mm
imp.reload(mm) # now if you've been fiddling with the functions
               # between runs of the main programme, the changes
               # will be picked up.

output_1, output_2 = mm.function(argument_1, argument_2, ... , argument_n)
```

That should make your life a lot easier and enable you to write much tidier main programmes!

6 Plotting

Python makes much nicer plots than IDL! I can guarantee it. First you need to

```
import matplotlib.pyplot as plt; plt.ion()
```

... which puts plotting in interactive mode, then make use of the following, which is far from exhaustive (Google is your friend), but covers a lot of stuff which I've had to do recently.

```
plt.plot(x,y)                # line plot
plt.plot(x,y,'r.')           # red dots
plt.plot(x,y,'r-',linewidth=2) # red line, double thickness
plt.plot(x,y,'r',ls='steps')   # ls = linestyle, this one gives
                               # a histogram style (for prebinned data)

plt.hist(x,bins=N,histtype='step') # bins the data x for you and plots
                                   # it in N bins (or you can use an
                                   # array N defining the bins wanted)

plt.scatter(x,y)                  # scatter plot
plt.errorbar(x,y,yerr=e,fmt='r.') # note the different call to get
                                  # the formatting you want.

plt.title('title')
plt.xlabel('x-axis label')
plt.legend([<names for the different sets of data>])

# more advanced:

plt.plot(x,y,'-',c=plt.cm.<chosen_colour_map>(value 0...255) )

# c= can take 'k', 'black', or something like (0.,0.,1.,1.) which is
# an RGBA tuple - the colour maps just point plot towards one
# of these.

# even more advanced:

fig,ax = plt.subplots(1,2) # subplots, one row, two columns

ax[0].plot( <the usual> ) # normal plotting works
ax[1].plot( <the usual> )

fig,ax = plt.subplots(2,2) # subplots, two rows, two columns

ax[0,0].plot( <the usual> )

# also

plt.axvspan(a,b,facecolor='r',alpha=0.5) # plots a red vertical bar
                                         # between x=a and x=b, which
                                         # is half-transparent
                                         # (axhspan does the same
                                         # for the other axis).

plt.cla() # wipes the plot from the figure
plt.clf() # wipes the figure from the window

# and finally, some 2D plotting options...

plt.contour(2D_ARRAY)  # contour plot
plt.contourf(2D_ARRAY) # filled contour plot

plt.contourf(X,Y,2D_ARRAY) # filled contour plot with axes given by
                           # X and Y

plt.contourf(X,Y,2D_ARRAY,cmap=plt.cm.<chosen_colour_map>({0...255})

plt.imshow(2D_ARRAY) # for a different style more suited to images

plt.imshow(2D_ARRAY,extent=(x0,x1,y0,y1),vmin=0,vmax=1400))

# additional commands refer to cuts in coordinates (extent) and pixel
# value (vmin,vmax), i.e. the limits of the colour scheme. cmap command
# can be used as with contours.
```

```
plt.colorbar() # puts a colour bar on the plot
```

The plotting window is interactive, it allows you to rescale the plot, or save it to disk (which is probably a big improvement on getting output to a file in IDL!).

7 Conclusion

... and that's it! Obviously it's not the last word, but this is all the stuff I found most useful during the first months of using Python, which means that if your work is anything like mine then the above will see you through most days.