Beginning Python

Table of Contents

Introduction

Python is a very popular language used in a variety of fields. However, it has only been adopted by the astronomy community on a large scale relatively recently. Thus, there are less astronomy libraries in python than, for example, IDL. However, since there are more code developers that work with python, the number of **libraries** of useful software is increasing dramatically.

Libraries are essentially bunches of add-on codes written in a language and people can use. Usually these codes have a unifying purpose. For example, someone might make a library of code that has to do with astro statistics, or one that does cosmological calculations. The possibilities are endless!

For the purposes of this class, we will be starting simple, but don't be surprised if you end up using some user-made libraries for your research projects (your mentors will help you, or course). In other words, what we do here is just the beginning!

The Interactive Environment and Basic Operations

Ok, first thing's first. Python has an *interactive environment*, which is pretty convenient to use. Simply type `ipython` into your terminal. You will get something that looks like this:

```
% ipython
Python 2.7.5 |Anaconda 1.6.1 (64-bit)| (default, Jun 28 2013,
22:10:09)
Type "copyright", "credits" or "license" for more information.
IPython 0.13.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
In [1]:
```

Note: You can also type the command `python` and you will get a slightly different (and MUCH less cool) interactive environment. I see no reason to ever use this.

Interesting Fact: When using `ipython`, you can do things that would normally be reserved for the command line like `cd` and `ls`. Try it out! Pretty cool!

To exit python, just type "exit" and hit return.
Now you are ready to start doing stuff!

Some Basic Operations, Understanding Variable types

Python Statements	Notes and Questions
`> print 3*5` `> 3*5`	1. What happens if you use a comma? Or if you don't even say "print"?
`> a = 3*5`	*This equal sign is an assignment. Read this as "a gets the value of 3 times 5."*
`> type(a)` `> type(A)`	3. What does type do? What's the difference between using a and A? *"INT" means integer.*
`> d=32767` `> d+1`	*Integers run from -32768 to + 32768.* 4. What happened?
`> d+1.`	5. What is the difference between using d+1 and d+1.?
`> x = d+1.` `> type(x)`	*Float: Floating-point, with six significant places.* *Double: Floating-point with approximately sixteen decimal digits.* *String: A sequence of 0 to 32,767 characters.*
`> 3/5` `> 3/5.`	What is the difference? *One floating number makes the result a float.*
`> width = 20` `> height = 5*9` `> width * height`	6. Explain what happened in these three lines.
`> print 2**2` `> a = 5` `> print 2**a, 2**2**a`	7. What does the `**` do?

Importing Modules and using Functions

Most functionality in python requires the use of modules that you must **import** before using. The Syntax for importing modules is simple

```
> import <module name>
```

Modules will include useful functions (this is probably the reason you are importing it in the first place!). Functions in python are called in a way similar to the syntax in your math classes. A function called "`Function`" that takes an input, `x` is called by writing `Function(x)`. If `Function` takes more than one variable, then it is called by writing `Function(x,y,z,...)`.

To use a function from the module called `module` you do the following:

```
> import module
> module.function(x)
```

For example:

```
> import math
> math.cos(0)
```

If you know you want to use all the functions in a module you can make your life a little bit easier.

```
> from math import *
> cos(0)
```

Now you don't have to write `math` before every function you use! Why would you **NOT** want to do this? Different modules may have functions with the same name. If you do this, the last module function you import will ***overwrite*** the previous module functions of the same name. Be very careful!

Some modules have sub-modules that you can get separately.

```
> from module import submodule
> submodule.function(x)
```

You can also change the name of a module for your own purposes. This is nice when a module is used a lot, but you don't want to use the import * command.

```
> import module as mod
> mod.function(x)
```

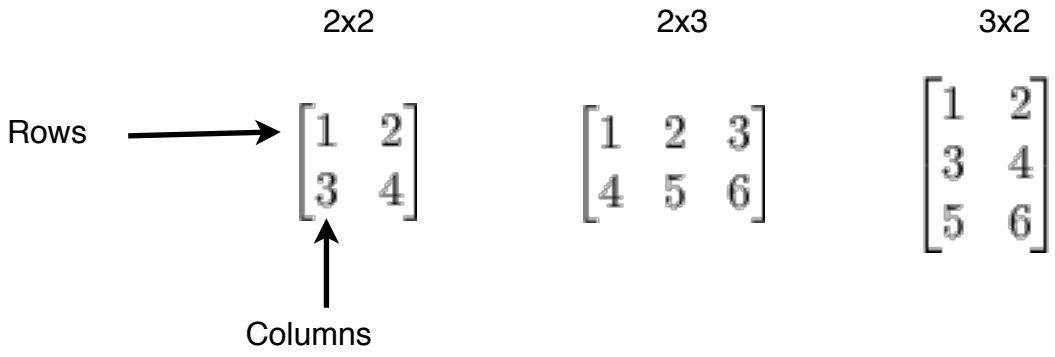
Using Some Math Functions

Python Statements	Notes and Questions
`> import math` `> print math.cos(180), math.cos(90)`	8. Is this in degrees or radians?
`> math.pi`	*math.pi is an **internal variable** in the math module. Python knows what you mean without you having to define it! (as long as you have loaded "math")*
`> print math.cos(math.pi), math.cos(math.pi/2)` `> print math.sin(math.pi), math.tan(math.pi/2)` `> from math import *` `> pi` `> cos(pi)`	*Aha! Much better!* *Notice the difference in syntax between regular import and from __ import **|
`> print acos(0), asin(0)` `> print acos(0)*180/pi` `> print asin(0)*180/pi` `> print acos(1), asin(1)` `> print acos(1)*180/pi` `> print asin(1)*180/pi`	9. What do **acos** and **asin** do? What does multiplying by 180/pi do?
`> a = 5` `> print a, log10(a), log10(10^a)` `> print exp(a), log(exp(a))`	10. What does **alog10()** do? 11. What do **exp()** and **alog()** do?
`> print 10 % 4` `> print 10 % 3` `> print 9 % 3` `> print 10 % 2, 10 5 5`	12. What does "**%**" do? Hint: think about division and *remainders!*

Working with Arrays

Arrays are convenient ways to store data. Every array is essentially a *matrix* of values.

Recall what a matrix is:



You can also have one dimension matrices (i.e. 3x1, 5x1, etc. **Vectors** are examples of 3x1 arrays!). You can also have more dimensions more dimensions (though those are harder to visualize)

Each value is given a particular *address* based on where it is in the matrix. Take this matrix, for example:

a is in position 1,1
b is in position 1, 2
f is in position 3, 2

$$\begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$$

Note that all of the "addresses" above have two numbers. This is because the matrix is a *two dimensional* matrix. If it was, say, 5 dimensional then it would have 5 numbers in its "address"

In Python, we need a special module to use arrays. This module is called `numpy` and it is one of the most useful tools that exists in python!

To use it, we just import it like any other module.

```
> import numpy as np
```

This allows us to just type 'np' instead of 'numpy' when we want to use the module.

Defining And Manipulating Arrays in Python

Python Statements	Notes and Questions
`> import numpy as np` `> a =  np.zeros(5)` `> a` `> a = np.zeros(5,dtype=int)` `> a` `> b = np.zeros(5,dtype=float)` `> b` `> c = np.zeros((5,3))` `> c` `> d = np.zeros((5,4,3))` `> d`	**np.zeros** *allows you to define arrays filled initially with zeroes. The default is always floats, but you can change it by specifying* dtype *You will need to get used to working with arrays of more than two dimensions. Look at how python "visualizes" these arrays. Try to make a 4 dimensional array; a 5 dimensional array. The visualization is not so good now...*
`> a = np.zeros(100)` `> print a` `> a` `> a = a +2` `> 10**a` `> a**2` `> import math` `> math.exp(a)` `> np.exp(a)` `> b = np.exp(a)` `> b`	13. What is the difference between using **print** and not using it? *Now we are manipulating arrays! Here is why arrays are so useful. They allow you to do the* ***same operation on an entire set of data.*** *Inputting an array into a function gives you a new array....* ***as long as you use the instance of the function that is in numpy!*** ***This also illustrates why doing*** `from ___ import *` ***is dangerous... math and np have many of the same functions!***

`> print a[0], a[2]` `> a[0] = 10` `> a[10]` `> c = [0,3,5,2]` `> a[c]` `> a[0], a[3], a[5], a[2]` `> print c[11]`	*You can also very easily select parts of data from arrays, whether it be a single element or a range of elements. This is called indexing an array.* ***Note: array counting starts at 0*** *You can also index an array with another array. It will output the elements of the array in the order given to it. Notice what happens when we try to index an array with a number greater than its size.*
`> a = np.arange(100)` `> a[5:]` `> a[1:5]` `> a[1:10:2]`	14. What does np.arange do? *Indexing in python can be complicated... but that is because you can do so much!* 15. What do each of these indexing commands do? *a[start:finish:by] will output the elements of a starting with index "start", ending at index "finish" going every "by" indices*
`> b = np.zeros((10,10,3))` `> b` `> b[0,0,0] = 10` `> b` `> b[:,0,0] = 1` `> b` `> b[0:4,0,0] = 2` `> b` `> b[1,0:4,0] = 3` `> b`	*Now onto higher dimensions....* 16. Explain what each one of these commands is doing.
`> a = np.zeros(10)` `> b = np.zeros((10,10))` `> a[0] = 2` `> a[1:9] = 3` `> b[0,:] = 2` `> b[0,:]*a` `> c = np.zeros((3,5))` `> print b*c`	*Arrays can be multiplied together!* ***Make sure you can explain, in words, what each of these commands do*** 17. What happens when you multiply arrays with **different** dimensions? Multiplying arrays that have different sizes is usually not a good idea.

Defining and Manipulating Arrays with Numpy.

Python Statements	Notes and Questions
`> a = np.zeros((3,5))` `> a` `> np.transpose(a)`	18. What does **transpose** do?
`> b = np.arange(100)` `> print np.size(b), np.size(a)` `> b = b + 3` `> np.where(b > 10)` `> np.where(b < 10)` `> np.where(b == 11)` `> o = np.where(b < 10)` `> b[o]` `> b[np.where(b<10)]` `> b[(b<10)]` `> np.where((b>5) & (b<50))` `> np.where((b==5) | (b==10))` `> np.where(b==1000000)`	19. What does where do? what do '<', '>' and '==' indicate? *You can use the where statements to index an array! In fact, you don't even need to have the actual "where" statement!* *The "&" essentially just means "and" like you expect. The "|" means "or"* *An <u>empty array</u> as a result means that there were no values that matched your criteria*
`> c = np.array([1,6,3,4,9,8,7])` `> c` `> np.sort(c)` `> print np.max(c), np.min(c)` `> d = [3,5,2,7,4,8,10]` `> print, d[(c == np.max(c))]`	This is another useful way of defining arrays. Just get a list of numbers in brackets and use the "np.array" function to turn it into an array type. 20. What does "sort" do? 21. What does np.min/max do? 22. explain what is happening in that last line.

If Statements and For/While Loops

"If" statements allow you to tell python to do something only if a certain logical statement is true.

The Syntax is rather simple.

```
> if [logical true/false statement]:[do something]
```

The logical statements used in "if" statements will have similar syntax to the "where" statements used above. Examples of good logical statements would be:

```
> if var < 0: print "hello"
> if var > 0: print "hello"
> if var == 0: print "hello"
```

Assuming that "var" is just some variable (NOT an array)

Your "if" statement must be a statement that is true or false. So, you have to be careful using arrays. For example, the following would cause problems.

```
> a = np.array([1,2,3,4,5,6])
> if a == 6: print "hello
```

The above if statement is both true and false, since one element of a is equal to 6. In other words, it makes no sense and python will yell at you!

```
> if a < 10: print "hello"
```

Even though all of the elements in a are less than 10, python still won't like that you are leaving the possibility for both true and false statements.

Of course, there are ways to use arrays in "if" statements. For example:

```
> if np.size(a)==6: print "hello"
```

"For" and "While" Loops

Sometimes, you want to run a piece of code many times. For this, "for" and "while" loops are very useful.

The general syntax for a for loop is as follows

```
> for i in [list/array of values]: [do something]
```

Python sets `i` equal to the first value in your list, then runs the piece of code within the loop. It then sets i equal to the next value and runs the loop again, continuing until there are no more values in your list.

To loop through a range of numbers starting at 0 do the following.

```
> for i in range(10): print i
```

The output from the above statement will be:

```
0
1
2
3
4
5
6
7
8
9
```

You can also have lists defined like this:

```
> for i in [1,8,10,3]: print i
```

Or, you can use arrays!

```
> x = np.array([1,9,3,10])
> for i in x: print i
```

While loops, on the other hand, will loop continuously until the "while" statement is no longer met. In other words, you should give python a logical statement that is initially true, but will become false eventually. For example:

```
> a = 0
> while a < 10: a = a +1
> print a
```

What happened above? First, we initialized the variable "a" to be 0, which is obviously less than 10. So, the while loop kept adding 1 to our variable. After each iteration, it checks to see if it is still less than 10. If it is, then it adds another 1 to it. Once a = 10, python checks once more and the logical statement is now false, so it stops. At the end of the loop, a = 10.

We can combine our loops with if statements. For example:

```
> a = np.zeros(100)
> a[55] = 1
> for i in range(100): if a[i] == 1 then print i
```

What happened above? We first initialized “a” to be an array where one of the elements was 1 and the rest 0. We then loop through the elements of the array using “i” . For each of these elements, we check to see if it equals 1. If it does, then we print out the value of “i”.

Loops will get much more interesting once we start making our own functions and procedures. In Python you can do multiple line loops and “if" statements within the interactive environment. Just hit return after each line! You will see a “. . . . :” show up. This tells you that your lines are going to happen within the loop. Notice also the tab structure that python gives you automatically... this will be important later… When you are done with your loop, hit return twice.

```
> x = 0
> for i in range(10):
....:     x = x + 1
....:     if x<10: print x
....:     y = 0
....:     while y < x:
....:          y = y+1
....:     print "y:", y
....:
```

Be able to explain what the above piece of code does!

File input/output

Most of the time, you will be working with data. That data is going to be located in files. Thus, reading files is a must! Luckily it is (relatively) straight forward.

Often, the files you read will be columns of data, where each column represents a certain type of data and each row represents a different object (like the BrightStars.dat file we worked with already).

The goal is to make a series of arrays, one for each column, where the i^{th} element in each array corresponds to the i^{th} object in the file.

Reading in Data
Method One: The more "traditional" way

First, you need to open the file!

```
> file1 = open("filename",'r')
```

The "r" means we are opening the file specifically to read it.
To read all the lines in the file, you will need to make a loop.

```
>  content = file1.readlines()
```

This command will make a list (slightly different from an array… don't worry too much about that), where each element is a line in file1. Each line is treated as a singe string.

```
> content[0].split([delimiter])
```

The split command will make an array of strings from a single line, where the elements are separated by the [delimiter] character. The default delimiter is a space. Other common delimiters are commas, colons, semi-colons, etc... though spaces are the most common by far. How do we turn this into an array with actual numbers?

First, define your arrays. Say we have N columns in our file. Define N arrays of the same size as there are lines in the file.

```
> var1 = np.array([])
> var2 = np.array([])
...
> varN = np.array([])
```

Now, we want to loop through each line and set the elements to the correct arrays.

```
> for i in range(np.size(content)):
     tmp = content[i].split()
     var1 = np.append(var1,tmp[0])
     var2 = np.append(var2,tmp[1])
     ...
     varN = np.append(varN,tmp[N-1])
```

Now we have a bunch of string arrays for each column of data.

```
> var1 = var1.astype(int)
> var2  = var2.astype(float)
....
> varN = varN.astype(int)
```

Again, we use the astype command to change the variable type of each array to match the individual pieces of data.

Method Two: The Simple Way

First, you will need to copy over a very useful file.

```
% cp ~premapta/python/readcol.py ~/python/pro
```

Now you can open the file readcol in python. You can take a look inside readcol.py to see the different options and what they do.

```
> import readcol
> var1,...,varN = readcol.readcol('filename',twod=False)
```

The above command will make N one-dimensional arrays. You can also put the data in a single two dimensional array of size MxN where M is the number of lines in your file and N is the number of columns (note that you have to make `twod=True`).

```
> data = readcol.readcol('filename',twod=True)
```

The greatest thing about readcol.py is that this code is not only fast, but it is able to correctly guess the type of variable in each column. It can also do some cool things if the column names are given at the top of the file. Again, look in readcol.py for more info on options.

There are several other options for reading files, such as np.genfromtxt and np.loadtxt. They all have advantages and disadvantages. The main disadvantage of these is that they cannot handle files with different data types (e.g. strings and floats)

Writing To Files

Writing to files is simple. Again, first you must open the file, this time for writing.

```
> file = open('filename','w')
> file.write("stuff")
```

Note: python will not go to a new line every time you do the "write" command. To create a new line, use the "\n" at the end of your string.

```
> file.write("stuff\n")
```

Don't forget to close the file!

```
> file.close()
```

Note: opening a file with "w" will overwrite your stuff! If you want to add on to a file, use the "a" option, which will open the file for appending.

```
> file = open('filename','a')
```

Writing Your Own Executables and Functions

You can save python commands in executable scripts. Open up a file ending in ".py" with emacs.

```
% emacs pythonScript.py
```

Put in all the code you want to run in the .py file. Even importing modules can be done within the file.

The executable can be run from the command line like this:

```
% python pythonScript.py
```

Python will run everything in the file in order.

Alternatively, what you can do is run the file within the iPython environment.

```
> import pythonScript
```

Again, this will run the scrip from beginning to end.

Note: When writing for/while loops or if statements in .py files, you must use spaces/tabs correctly. Code within each loop must have the same amount of "whitespace" before it.

```
for i in range(100):
<tab>[do things]
<tab>for j in range(100):
<tab><tab>[do things]
<tab><tab>[do more things]
<tab>[do some other stuff]
<tab>if [true/false statement]:
<tab><tab>[do stuff]
```

Defining Functions

You can define functions within .py files, a very useful way of organizing your code.

Functions are defined like this:

```
def func(var1,var2,...,varN):
     [do stuff]
     [do more stuff]
     [everything must be tabbed in once!]
     for i in range(100):
          [you can even do loops!]
          [but be careful of tabs/spaces]
```

The above defines a function called `func` that takes N inputs called `var1,var2,...` You don't have to have anything ending the function. Just go to the next line without tabbing. For example:

```
[import all the stuff you need first]

def func1(x,y,z):
     [stuff]

def func2(x,y,z):
     [other stuff]

ect...
```

Now, you can use these functions within a script within the .py file and then run it using the python filename.py command. Or, you can utilize them in the interactive environment (basically, think of your script as another **module** to import!)

```
> import pythonScript.py
> pythonScript.func1(x,y,z)
> pythonScript.func2(x,y,z)
```

Functions can return values to you. To do this, just use this syntax:

```
return value1, value2,value3
```

The above line will a return 3 numbers. To store these values, just do the following:

```
> a,b,c = pythonScript.func1(x,y,z)
```

a,b,c will be set to value1,value2,value3 respectively.

```
> list = pythonScript.func1(x,y,z)
```

If you just set the output of `func1` to one variable, you will get a **list**. A list is somewhat like an array, but with less functionality. You can index a list similar to a one dimensional array.

```
list = (a,b,c)
list[0] = a
list[1] = b
list[2] = c
```

Intro to Plotting

For plotting, we use matplotlib, which is a *very* useful library. In particular, we will use the sub-module pyplot within matplotlib.

Python Statements	Notes and Questions
`> import numpy as np` `> import matplotlib.pyplot as plt` `> plt.ion()` `> x = np.arange(100)` `> y = np.arange(100)/50.` `> plt.plot(x,y,'r-')` `> plt.clf()` `> plt.plot(x,y,'-')` `> plt.plot(x,y/2,'--')` `> plt.plot(x,y*2,'.-')`	*This line puts you in interactive mode* *"plt.plot" will open up a window with your plot in it. The string at the end sets the style and color of the line. In this case, r = red, - = line.* *Close the current plot window and start over.* *if you plot more data, it will show up on the same plot. If you don't specify a color python will automatically make it a color and it will choose a different color for each set of data you plot.* *Some other useful strings:* *b = blue ; k = black ; g = green ; -- = dashed line* *+ = '+' symbol ; 'x' = 'x' symbol ; . = dots* *.- = dot-dash ; o = big dots*
`> plt.xlabel('x-axis')` `> plt.ylabel('y-axis')` `> plt.title('This is a Plot')`	*You can title your plot and label the axes.*
`> plt.axis([0,50,0,50])` `> plt.axis([100,0,0,50])`	*You can also change the axis ranges. The general format is* `plt.axis([xmin,xmax,ymin,ymax])` You can even reverse the axis by switching xmin and xmax.
`> plt.savefig('name.pdf')` `> plt.close()`	you can save the current figure to a file. Use '.pdf' or '.png' for the best results.
`> plt.subplot(321)` `> plt.plot([0,1],[0,1])` `> plt.subplot(323)` `> plt.plot([0,1],[0,1])` `> plt.close()`	*This will make a 3 by 2 set of plots. The third number selects your current working plot (1 through 6 in this case)* 23. What is the difference between plt.close and plt.clf?

Making legends in Python

This is super easy. When you plot anything (even a histogram) just add in the keyword `label = "whatever you want to call these data points/lines"`. Then, when you have plotted everything you need, type in the command `plt.legend()` and python will make your legend!

The only really important keyword for legend is `loc = "string"`
This will set the location of your legend. The strings you can use that will be understood are: 'best', 'upper right', 'upper left', 'lower right', 'lower left', 'right', 'center', 'center left', 'center right', 'lower center', 'upper center', 'center'

Hopefully those are pretty self-explanatory! If you are working in the plt.ion() environment, you can actively experiment and see the results in real time!

Histograms

Histograms are relatively simple things to make in python. We will use matplotlib. The function we will use is `plt.hist` which has the following syntax

```
> n,bins,patches = plt.hist(intputarray, bins=100)
```

This will not only plot the histogram for you, but it will return the number of elements in each bin (n) and the bins themselves (bins). You can name the three variables holding the output anything you want, but you need three. Don't worry about what the "patches" are.

Keywords for the hist() function

You can denote the color and linestyle in the same way as you do in plt.plot()

`bins`: number of bins you want to have. can also be a list of bin edges.
`range`: lower and upper range of the bins
`normed`: "= True" means you get a probability distribution instead of just raw number counts
`histtype`: 'bar' = traditional stype, 'step' = a line plot. looks better usually
`Weights`: this is an array of values that must have the same size as the number of bins you have. This will be a factor by which you will multiply the number count of each bin. In other words, it will make the "number of elements" output be n*weights instead. This is a good way to normalize your histogram outside of just using the normed variable. For example, if you wanted to plot the fraction of objects in each bin, you would set `weights` equal to an N sized array (N = number of bins you have) where each element of the array is equal to 1/(total # of objects).