Tauxe_HW_2

March 29, 2016

Lecture notes for Rock and Paleomagnetism, Spring Quarter, 2016, UC San Diego Lecture 2 “Python Programming Bootcamp”.
by Lisa Tauxe
Class website: http://magician.ucsd.edu/~ltauxe/sio247

0.1 Variable Types

The time has come to talk about variable types. We’ve been very relaxed up to now, because we don’t have to declare them up front and we can often even change them from one type to another on the fly. But - variable types matter, so here goes. Python has integer, floating point (both long and short), string and complex variable types. It is pretty clever about figuring out what is required. Here are some examples:

```
In [29]: import numpy as np
```

```
In [30]: number=1 # an integer
         Number=1.0 # a floating point
         NUMBER='1' # a string
         complx=1j # a complex number with imaginary part 1
         Complx=np.complex(3,1) # the complex number 3+1i
```

Try doing math with these!

```
In [31]: print number+number  #  [ an integer]
         print number+Number  # a float
         print NUMBER+NUMBER   #[a string]

2
2.0
11
```

But what about this!

```
In [32]: number+NUMBER # [Gives you an angry error message]


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-32-088a9d3a42c2> in <module>()
    ----> 1 number+NUMBER # [Gives you an angry error message]


        TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Lesson learned: you can't add a number and a string. and string addition is different! But you really have to be careful with this. If you multiply a float by an integer, it is possible that you will convert the float to an integer when you really wanted all those numbers after the decimal! So, if you want a float, use a float.

You can convert from one type to another (if appropriate) with:
int(Number); str(number); float(NUMBER);
long(Number); complex(real,imag)

long() converts to a double precision floating point and complex() converts the two parts to a complex number. There is another kind of variable called "boolean". These are: true, false, and, or, and not. For the record, the integer '1' is true and '0' is false. These can be used to control the flow of the program as we shall learn later.

0.2 Data structures

In previous programming experience, you may have encountered arrays, which are a nice way to group a sequence of numbers that belong together. In Python we also have arrays, but we also have more flexible data structures, like lists, tuples, and dictionaries, that group arbitrary variables together, like strings and integers and floats - whatever you want really. We'll go through some attributes of the various data structures, starting with lists.

0.2.1 Lists

- Lists are denoted with square brackets, [ ], and can contain any arbitrary set of items, including other lists!
- Items in the list are referred to by an index number, starting with 0.
- You can count from the end to the beginning by starting with -1 (the last item in the list), -2 (second to last), etc.
- Items can be sorted, deleted, inserted, sliced, counted, concatenated, replaced, added on, etc.

Here are a few examples of things you can do with lists:

```
In [61]: mylist=['a',2.0,'400','spam',42,[24,2]] # defines a list
         print mylist
         print mylist[2] # refers to the third item
         print mylist[-1] # refers to the last item
         mylist[1]=26.3   # replaces the second item
         print mylist
         del mylist[3] # deletes the fourth element
         print mylist
```

```
['a', 2.0, '400', 'spam', 42, [24, 2]]
400
[24, 2]
['a', 26.3, '400', 'spam', 42, [24, 2]]
['a', 26.3, '400', 42, [24, 2]]
```

To slice out a chunk of the middle of a list try this. It takes items 2 and 3 out (note it takes out up to but not including the last item number - don't ask me why).

```
In [62]: newlist=mylist[1:3]
         newlist
```

```
Out[62]: [26.3, '400']
```

Or, we can slice it this way which takes from the fourth item (starting from 0!) to the end:

```
In [63]: newlist=mylist[3:]
         print newlist
```

```
[42, [24, 2]]
```

To copy a list BEWARE! You can make a copy - but it isn't an independent copy (like in, e.g., Fortran), but it is just another name for the SAME OBJECT, so:

```
In [64]: mycopy=mylist
         mylist[2]='new'
         print mycopy[2]
```

```
new
```

See how mycopy got changed when we changed mylist? To spawn a new list that is a copy, but an independent entity, try:

```
In [65]: mycopy=mylist[:]
         mylist[2]=1003
         print mycopy[2]
```

```
new
```

So now mycopy stayed the way it was, even as mylist changed.

List methods Python is "object oriented", a popular concept in coding circles. We'll learn more about what that means later, but for right now you can walk around feeling smug that you are learning an object oriented programming language. O.K., what is an object? Well, mylist is an object. Cool. What do objects have that might be handy? Objects have "methods" which allow you to do things to them. Methods have the form: object.method()

Here are an example:

```
In [74]: mylist=['a',2.0,'400','spam',42,[24,2]]
         mylist.append('me too') # appends a string to mylist
         print mylist
```

```
['a', 2.0, '400', 'spam', 42, [24, 2], 'me too']
[2.0, 42, [24, 2], '400', 'a', 'me too', 'spam']
None
```

For a complete list of methods for lists, see: http://docs.python.org/tutorial/datastructures.html#more-on-lists

0.2.2 More about strings

Numbers are numbers. While there are more kinds of numbers (complex, etc.), strings can be more interesting. Unlike in some languages, they can be denoted with single, double or triple quotes: e.g., 'spam, "Sam's spam", or:

""""

Hi there - we can type as
many lines as we want!

""""

```
In [67]: """
         Hi there - we can type as
         many lines as we want!
         """
```

```
Out[67]: '\nHi there - we can type as\nmany lines as we want!\n'
```

Strings can be added together and sliced. But they CANNOT be changed in place - you can't do this: newstring[0]='b'. To find more of the things you can and cannot do to strings, see: http://docs.python.org/tutorial/introduction.html#strings

```
In [68]: newstring = 'spam' + 'alot'
         print newstring[0:3]
```

```
spa
```

0.2.3 Dictionaries!

Dictionaries are denoted by {}. They are also somewhat like lists, but instead of integer indices, they use alphanumeric 'keys': I love dictionaries. So here is a bit more about them.

To define one:

```
In [78]: Telnos={'lisa':46084,'lab':46531,'jeff':44707} # defines a dictionary
```

To return the value associated with a specific key:

```
In [79]: Telnos['lisa']
```

```
Out[79]: 46084
```

To change a key value:

```
In [80]: Telnos['lisa']=46048
```

To add a new key value:

```
In [81]: Telnos['newguy']=48888
```

Dictionaries also have some methods. One useful one is:

```
In [82]: Telnos.keys()
```

```
Out[82]: ['lisa', 'newguy', 'lab', 'jeff']
```

For a more complete accounting of dictionaries, see: http://docs.python.org/tutorial/datastructures.html#dictionaries

Arrays in Python have many similarities to lists. Unlike lists, however, arrays have to be all of the same data type (dtype), usually numbers (integers or floats), although there appears to be something called a character array. Also, the size and shape of an array must be known a priori and not determined on the fly like lists. For example we can define a list with L=[], then append to it as desired, but not so arrays - they are much pickier and we'll see how to set them up later.

Why use arrays when you can use lists? They are far more efficient than lists particularly for things like matrix math. But just to make things a little confusing, there are several different data objects that are loosely called arrays, e.g., arrays, character arrays and matrices. These are all subclasses of ndarray. I'm just going to briefly introduce arrays here.

Here are a few ways of making arrays:

```
In [88]: import numpy as np
         A= np.array([[1,2,3],[4,2,0],[1,1,2]])
         print 'A= ', A
         B=np.arange(0,10,1).reshape(2,5)  # guess what reshape() does!
         print 'B= ', B
         C=np.array([[1,2,3],[4,5,6]],np.int32)
         print 'C=', C
```

```
        D=np.zeros((2,3)) # Notice the zeros and the size is specified by a tuple.
        print 'd= ', D
        E=np.ones((2,4))
        print 'E= ', E
        F=np.linspace(0,10,14)
        print '= ', F
        G=np.ndarray(shape=(2,2), dtype=float)
        print 'G= ', G # note how this is initalized with really low numbers (but not zeros).
```

```
A=  [[1 2 3]
 [4 2 0]
 [1 1 2]]
B=  [[0 1 2 3 4]
 [5 6 7 8 9]]
C= [[1 2 3]
 [4 5 6]]
d=  [[ 0.  0.  0.]
 [ 0.  0.  0.]]
E=  [[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
=  [  0.          0.76923077   1.53846154   2.30769231   3.07692308
   3.84615385   4.61538462   5.38461538   6.15384615   6.92307692
   7.69230769   8.46153846   9.23076923  10.        ]
G=  [[ 0.  0.]
 [ 0.  0.]]
```

Note the difference between linspace(start,stop,N) and arange(start,stop,step). The function linspace creates an array with 14 evenly spaced elements between the start and stop values while arange creates an array with elements at step intervals between the starting and stopping values.

Python arrays have methods like dtype, ndim, shape, size, reshape(), ravel(), transpose() etc.

Here are some particularly handy examples:

```
In [87]: A= np.array([[1,2,3],[4,2,0],[1,1,2]])
         L=A.tolist() # converts array A to list L
         print 'L=',L
         A=np.array(L) # converts list  L to array A
         print 'A=',A
```

```
L= [[1, 2, 3], [4, 2, 0], [1, 1, 2]]
A= [[1 2 3]
 [4 2 0]
 [1 1 2]]
```

For a lot more tricks with arrays, go to the NumPy Reference website here: http://docs.scipy.org/doc/numpy/reference/

See especially the rules for slicing and dicing:

http://docs.scipy.org/doc/numpy/reference/arrays.indexing.html

0.3 Pandas

Now that you have fallen in love with Python and Numpy, we have a new treat for you: Pandas. It supports elegant data structures and tools for wrangling the data which allow fast and user-friendly data analysis. There are two basic data structures: the Series (a one-dimensional array like object that is a data array with an associated array of indices which can be numbers or text) and the DataFrame (a spreadsheet like data table).

First, invoke Pandas by importing it, then create some data Series using several approaches:

```
In [93]: import pandas as pd
         # create first pandas object with numeric indices
         obj = pd.Series([1,2,3,4])
         print obj

0    1
1    2
2    3
3    4
dtype: int64
```

```
In [97]: # create second pandas object with text indices
         obj2=pd.Series([1,2,3,4],index=['a','b','c','d'])
         obj2

Out[97]: a    1
         b    2
         c    3
         d    4
         dtype: int64
```

```
In [99]: # create a third pandas Series from the dictionary
         obj3=pd.Series(Telnos)
         obj3

Out[99]: jeff      44707
         lab       46531
         lisa      46048
         newguy    48888
         dtype: int64
```

Another very useful pandas object is the DataFrame. These have both row and column indices and are like a dictionary of Series so are much more flexible than the Numpy array objects. Here is an example:

```
In [102]: data={'telnos':['44707','46531','46084'],\
              'names':['jeff','lisa','lab'],\
              'rooms':['300D RH','300E RH','1162 SvH']}
          frame=pd.DataFrame(data)
          frame

Out[102]:   names     rooms telnos
          0  jeff   300D RH  44707
          1  lisa   300E RH  46531
          2   lab  1162 SvH  46084
```

There are many methods associated with pandas objects which allow searching, massaging and generally fiddling with the data. You should check out some of the many tutorials on pandas on the web.

0.4 Code blocks

Any reasonable programming language must provide a way to group blocks of code together, to be executed under certain conditions. In Fortran, for example, there are if statements and do loops which are bounded by the statements if, endif and do, endo respectively. Many of these programs encourage the use of indentation to make the code more readable, but do not require it. In Python, indentation is the way that code blocks are defined - there are no terminating statements. Also, the initiating statement terminates in a colon. The trick is that all code indented the same number of spaces (or tabs) to the right belong together. The code block terminates when the next line is less indented. A typical Python program looks like this:

```
 program statement
block 1 top statement:
    block 1 statement
    block 1 statement          ha-ha i can break the indentation convention!
    block 1 statement
    block 2 top statement:
        block 2 statement
        block 2 statement
        block 3 top statement:
            block 3 statement
            block 3 statement
            block 4 top statement: block 4 single line of code
        block 2 statement
        block 2 statement
    block 1 statement
    block 1 statement
program statement
```

Exceptions to the code indentation rules are:

- Any statement can be continued on the next line with the continuation character and the indentation of the following line is arbitrary.
- If a code block consists of a single statement, then that may be placed on the same line as the colon.
- The command break breaks you out of the code block. Use with caution!
- There is a cheat that comes in handy when you are writing a complicated program and want to put in the code blocks but don't want them to DO anything yet: the command pass does nothing and can be used to stand in for a code block.

In the following, I'll show you how Python uses code blocks to create "do" and "while" loops, and "if" statements.

But first a few more basics:
len(mylist) returns the number of items in the list mylist * range(start,stop,increment) returns a list of integers from start to stop MINUS ONE, in increments of inrement.

0.4.1 For blocks

```
In [104]: mylist=[42,'spam','ocelot']
          for item in range(0,len(mylist),1): # len(mylist) is the length of mylist
             print mylist[i]
          print 'All done'
```

```
42
spam
ocelot
All done
```

Note that of course we could have used any variable name instead of 'item', but it makes sense to use variable names that mean what they do.

Here is an example with a little more heft to it. It creates a table of trigonometry functions, spitting them out with a formatted print statement:

```
In [106]: import numpy as np
          deg2rad = np.pi/180. # remember conversion to radians from degrees
          for theta in range(90): # short form of range, returns [0,1,2...89]
             ctheta = np.cos(theta*deg2rad) # define ctheta as cosine of theta
             stheta = np.sin(theta*deg2rad)# define stheta as  sine of theta
```

```
        ttheta = np.tan(theta*deg2rad)   # define ttheta as tangent of theta
        if theta<10:
            print '%5.1f %8.4f %8.4f %8.4f' %(theta, ctheta, stheta, ttheta)
```

```
0.0   1.0000   0.0000   0.0000
  1.0   0.9998   0.0175   0.0175
  2.0   0.9994   0.0349   0.0349
  3.0   0.9986   0.0523   0.0524
  4.0   0.9976   0.0698   0.0699
  5.0   0.9962   0.0872   0.0875
  6.0   0.9945   0.1045   0.1051
  7.0   0.9925   0.1219   0.1228
  8.0   0.9903   0.1392   0.1405
  9.0   0.9877   0.1564   0.1584
```

To make the output look nice, we do not use
print theta, ctheta, stheta, ttheta
which would space the numbers irregularly among the columns and put out really long numbers. Instead, we explicitly specify the output format. The output format is given in the quotes. The format for each number follows the %, 5.1f is for 5 spaces of floating point output, with 1 space to the right of the decimal point. The single blank space between %5.1f and %8.4f is included in the output, in fact any text there is reproduced exactly in the output, thus to put commas between the output numbers, write:
print ‘%5.1f, %8.4f, %8.4f, %8.4f’ %(theta, ctheta, stheta, ttheta)
Tabs (\t) would be formatted like this:
print ‘%5.1f’‘%8.4f’‘%8.4f,’ %8.4f’ %(theta, ctheta, stheta, ttheta)

0.4.2 If and while blocks

The “for block” is just one way of controlling flow in Python. There are also if and while code blocks. These execute code blocks the same way as for loops (colon terminated top statements, indented text, etc.). For both of these, the code block is executed if the top statement is TRUE. For the “if” block, the code is executed once but in a “while” block, the code keeps executing as long as the statement remains TRUE.

The key to flow control therefore is in the top statement of each code block; if it is TRUE, then execute, otherwise skip it. To decide if something is TRUE or not (in the boolean sense), we need to evaluate a statement using comparisons. Here’s a handy table with comparisons (relational operators) in Python:

- equals: ==
- does not equal: !=
- less than: <
- less than or equal to: <=
- greater than: >
- greater than or equal to: <=
- and: and
- or: or

These operators can be combined to make complex tests. Here is a juicy complicated statement:
if ((a > b and c <= 0) or d == 0):
code block
Use parentheses liberally - make the order of operation completely unambiguous even if you could get away with fewer.

0.4.3 Finer points of if blocks:

There are whistles and bells to the ‘if’ code blocks. In Python these are: elif and else. A code block gets executed if the top if statement is FALSE and the elif statement is TRUE. If both the top if and the elif statements are FALSE but the else statement is TRUE, then Python will execute the block following the else. Consider these examples:

```
In [112]: mylist=['jane','brian','denise']
          if 'maureen' in mylist:
              pass # don't do anything
          if 'maureen' not in mylist:
             print 'call maureen and apologize!'
             mylist.append('maureen')
          elif 'cathy'  in mylist: # if first statement is false, try this one
              print 'cathy also not on the list'
          else: # if both statements are false, do this:
             print "both maureen and cathy aren't in the list"
```

```
call maureen and apologize!
```

0.4.4 While loops

As already mentioned, the 'while' block continues executing as long as the while top statement is TRUE. In other words, the if block is only executed once, while the while block keeps looping until the statement turns FALSE. Here are a few examples:

```
In [116]: a=1
          while a < 10:
              print a
              a+=1
          print "I'm done counting!"
```

```
1
2
3
4
5
6
7
8
9
I'm done counting!
```

0.5 File I/O in Python

Python would be no better than a rather awkward graphing calculator (and we haven't even gotten to the graphing part yet) if we couldn't read data in and spit data out. You learned a rudimentary way of spitting stuff out already using the print statement, but there is a lot more to file I/O in Python. We would like to be able to read in a variety of file formats and output the data any way we want. In the following we will explore some of the more useful I/O options in Python.

You can read data in using the native Python function 'readlines()', the NumPy function 'np.loadtxt()' or the Pandas function 'pd.read_csv()'. Each of these has strengths and weaknesses. The function readlines() reads everything in as strings. loadtxt() reads things into arrays and has to be all of the same variable type (you can't mix numbers and strings). And read_csv() takes some getting used to (but is totally worth it).

Now we will create a data set, write it to a file and read it back in using all three methods.

0.5.1 Writing data to a file

You can read data in using the native Python function 'readlines()', the NumPy function 'loadtxt()' or the Pandas function 'read_csv()'. Each of these has strengths and weaknesses. The function readlines() reads everything in as strings. loadtxt() reads things into arrays and has to be all of the same variable type (you can't mix numbers and strings). And read_csv() takes some getting used to (but is totally worth it).

Remember the Pandas dataframe 'frame'?

```
In [126]: print frame

names     rooms telnos
0  jeff   300D RH  44707
1  lisa   300E RH  46531
2   lab  1162 SvH  46084
```

Let's write it to a tab delimeted file (filename 'data.txt'). The default column separator is a comma (hence the csv which stands for comma separated value), but many separators are possible. To specify a tab, use "sep='̂'". To do it without the sometimes annoying incices, specify index=False.

```
In [129]: frame.to_csv('data.txt',sep='\t',index=False)
```

You can look at your handy work with some text editor (Notepad, Textedit, vi!). Then read it back in with Pandas. To specify a header line, use header=0 for the first line, 1 for the second....

```
In [131]: newframe=pd.read_csv('data.txt',sep='\t',header=0)
          newframe

Out[131]:   names     rooms  telnos
          0  jeff   300D RH   44707
          1  lisa   300E RH   46531
          2   lab  1162 SvH   46084
```

0.6 Functions

So far you have learned how to use functions from program modules like NumPy. You can imagine that there are many bits of code that you might write that you will want to use again and again, say converting between degrees and radians and back, or finding the great circle distance between two points on Earth. The basic structure of a program with a Python function is:

```
In [133]: def FUNCNAME(in_args):
              """
              DOC STRING - says what it does!
              """
              print in_args
              out_args=42 # some code that does something
              return out_args # returns some stuff
          FUNCNAME('oh boy')

oh boy

Out[133]: 42
```

0.6.1 Finer points of functions

The first line must have 'def' as the first three letters, must have a function name with parentheses and a terminal colon. If you want to pass some variables to the function, they go where in_arg sits, separated by commas. There are no output variables here.

There are four different ways to handle argument passing.

1) You could have a function that doesn't need any arguments at all:

```
In [135]: def gimmepi():
              """
              Returns the value of pi
              """
              import numpy as np
              return np.pi
```

When you call it, you get some pi!

```
In [140]: pi=gimmepi()
          print pi
          # or just this:
          gimmepi()

3.14159265359

Out[140]: 3.141592653589793
```

2) You could use a set list of what are called 'formal' variables that must be passed:

```
In [142]: def deg2rad(degrees):
              """
              converts degrees to radians
              """
              return degrees*gimmepi()/180.

In [143]: print '42 degrees in radians is: ',deg2rad(42.)

42 degrees in radians is:  0.733038285838
```

3) You could have a more flexible need for variables. You signal this by putting *args in the in_args list (along with any formal variables you want):

```
In [146]: def print_args(*args):
              """
              prints the argument list
              """
              print 'you sent me these arguments:'
              for arg in args:
                  print arg
          print_args(1,4,'hi there')

you sent me these arguments:
1
4
hi there
```

4) You can use a keyworded, variable-length list by putting **kwargs in for in_args:

```
In [151]: def print_kwargs(**kwargs):
              """
              prints a keyworded list of arguments
              """
              for key in kwargs:
                  print '%s %s'%(key,kwargs[key])

In [152]: print_kwargs(arg1='NI!',arg2=42,arg3='ocelot')

arg1 NI!
arg2 42
arg3 ocelot
```

0.6.2 Doc string

Although you can certainly write functional code without a document string, make a habit of always including one. Trust me - you'll be glad you did. This can later be used to remind you of what you thought you were doing years later. It can be used to print out a help message by the calling program and it also let's others know what you intended. Notice the use of the triple quotes before and after the documentation string - that means that you can write as many lines as you want.

0.6.3 Function body

This part of the code must be indented, just like in a for loop, or other block of code.

0.6.4 Return statment

You don't need this unless you want to pass back information to the calling body (see, for example print_kwargs() above). Python separates the entrance and the exit. See how it can be done in the gimme_pi() example above.

0.6.5 Main program as a function

It is considered good Python style to treat your main program block as a function too. (This helps with using the document string as a help function and building program documentation in general.) In any case, we recommend that you just start doing it that way too. In this case, we have to call the main program with the final (not indented) line main():

```
In [155]: def main():
              """
              calls function print_kwargs
              """
              print_kwargs(arg1='NI!',arg2=42,arg3='ocelot') # defined above
          main() # runs the main program
```

```
arg1 NI!
arg2 42
arg3 ocelot
```

0.7 Build your own module

Notice how in the above examples, all the functions preceded the main function. This is because Python is an interpreter and not compiled - so it won't know about anything declared below as it goes through the script line by line. On the other hand, we've been running lots of functions and they were not in the program we used to call them. The trick here is that you can put a bunch of functions in a separate file (in your path) and import it, just like we did with NumPy. Your functions can then be called from within your program in the same way as for NumPy.

So let's say we put all the above functions in a file called myfuncs.py

Put this into a text file in your homework directory: you can copy it and then use the unix (or dos) command 'cat > myfuncs.py' (remember the ^D at the end).

```
In [ ]: def gimmepi():
            """
            returns pi
            """
            return 3.141592653589793
        def deg2rad(degrees):
            """
            converts degrees to radians
            """
            return degrees*3.141592653589793/180.
        def print_args(*args):
            """
            prints argument list
            """
            print 'You sent me these arguments: '
            for arg in args:
                print arg
```

```
In [156]: import myfuncs as mf
```

```
In [157]: mf.gimmepi()
```

```
Out[157]: 3.141592653589793
```

0.8 Matplotlib

So far you have learned the basics of Python, and NumPy. But Python was sold as a way of visualizing data and we haven't yet seen a single plot (except a stupid one in the beginning). There are many plotting options within the Python umbrella. The most mature and the one we am most familiar with is matplotlib, a popular graphics module of Python. Actually matplotlib is a collection of a bunch of other modules, toolkits, methods and classes. For a fairly complete and readable tour of matplotlib, check out these links: http://matplotlib.sourceforge.net/Matplotlib.pdf and here: http://matplotlib.sourceforge.net/

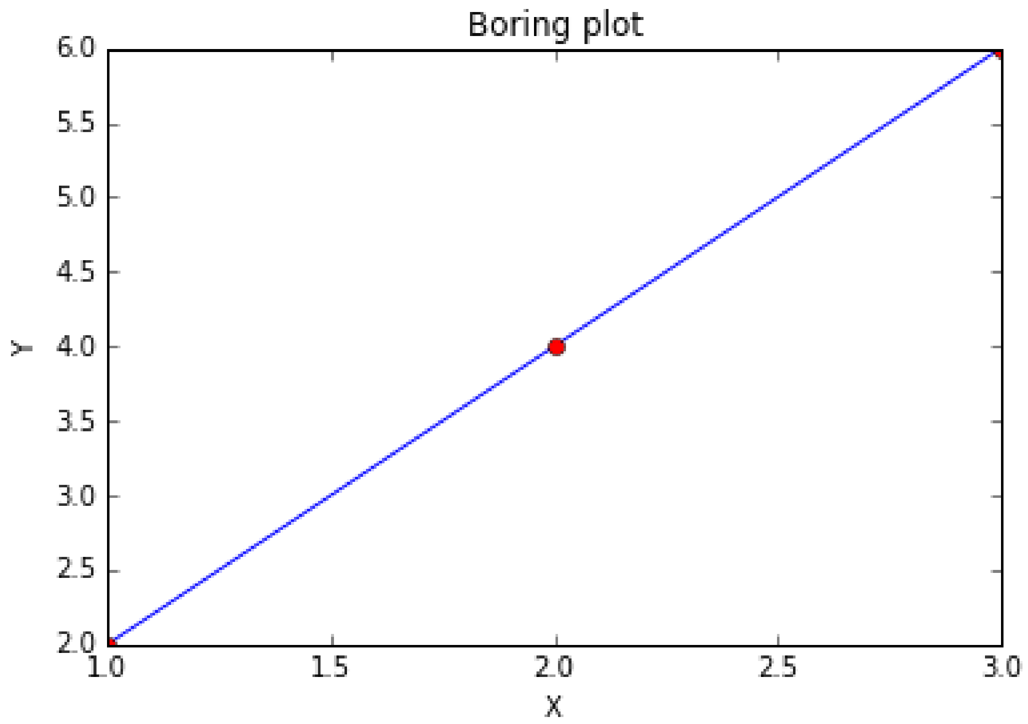
0.8.1 A first plot

Let's start by reviewing a simple plot script. But first we must import a few more modules to let us plot within the notebook.

```
In [161]: import matplotlib
          import pylab as plt
          %matplotlib inline
```

```
In [164]: X=[1,2,3] # define x variables
          Y=[2,4,6] # define y variables
          plt.plot(X,Y)# makes a line
          plt.plot(X,Y,'ro') # makes some little red circles
          plt.xlabel('X') # prints the X axis label
          plt.ylabel('Y') # figure it out yourself!
          plt.title('Boring plot') # ditto
```

```
Out[164]: <matplotlib.text.Text at 0x112157950>
```



Of course there is lots more to learn about Python! We will embellish your skills in the next homework problems and of course there is abundant material online to learn from.

For your homework for this class. Download this notebook at: http://magician.ucsd.edu/~ltauxe/sio247/Lectures/lecture02.ipynb

Play with it. Run each cell block as you go or try to run them all at once (figure out how to debug the problem that you encounter doing that.) The next homework will be to write an actual original notebook with markdown and code so get ready!

```
In [ ]:
```