OADnotesday2

Unknown Author

September 03, 2013

1 Day2

Loops and control flows

Control flows allows us to get multiple outcomes when running a piece of code.

Boolean algebra

Usually control flows statements are connected by Boolean operators in a grammatically correct way—almost exactly as in regular English. There are three boolean operators in Python:

- and, which means the same as it does in English;
- or, which means "one or the other **OR BOTH**" (it's not exclusively one or the other, the way it often is in English);
- not, which means the same as it does in English.

And

The operator *and* is only true if and only if the expressions on each sides of it are **True** for e.g

- 1 < 2 and 2 < 3 results in **True** because it is true that 1 is less than 2 and that 2 is less than 3.
- 1 < 2 and 2 > 3 results in **False** because it is not true that both statements are true - 1 is less than 2, but 2 is not greater than 3.
- 4 > 5 and 6 < 2.5 results in **False** because it both statements are **False**.

Or

The boolean operator *or* only returns **True** when either (meaning one, the other or both!) of the expressions on each side of *or* are **True**. (It's only **False** when both expressions are **False**.) For e.g:

- 1 < 2 or 2 > 3 is **True**, even though two is not greater than three;
- 1 > 2 or 2 > 3 is **False**, because it is neither the case that one is greater than two nor that two is greater than three.

Not

The boolean operator *not* returns **True** for **False** statements and **False** for **True** statements! Remember, the only two boolean values are **True** and **False**! For example: not **False** will evaluate to **True**, as will not 40 > 41. Applying not to expressions that would otherwise be **True** makes them **False**.

If your friend ask you that on movie night you are going to watch either Starwars or Lord of the Rings, then he most probably means that you will watch only **one** of those 2 movies. But in python, in the mearning of **or**, the statement will still be valid (True) if you watch both movies!

Conditional Syntax

The Conditional Syntax that are present in Python are *if*, *elif* and *else* Usually the *if* statement has to be followed by an expression to be evaluated by the computer. E.g

```
If a > b:
    print "Hi!"
    print "bla bla bla"
    z = a*b
else:
    print 'Bye!'
    z = a-b
```

In the case of the above code the computer 1st evaluate if a is bigger than b. if a is bigger than b, then the computer goes on to evaluate the print statements and evaluate z = a x b. In this case the computer will completely **ignore** the statements inside the else indentation. In the case that a is smaller than b, then the computer directly skips to evaluate what is inside the else indentation.

Multiple if statements and elif

Sometimes there are multiple solution that could arise from a problem and you might want to evaluate something different for each of the cases. lets consider

```
if a > 0:
    print "Hey a is positive!"
if a > 100:
    print "Hey a is bigger than 100!"
if a < 0:
    print "Hey a is negative!"
if a == 0:
    print "a is zero"
```

In the case above, the computer will evaluate **EACH** of the *if* statements to check if they are **True** or **False**. Say we set a = 123, then the computer will print both “Hey a is positive” and “Hey a is bigger than 100!”. Since we specified all possibilities of a (i.e a>0, a<0 and a==0) then there is no need to use the else statement.

The elif statement is somewhat different. It is only executed if the statements above it are **False** -> if one of the statement above it is **True** then it will not be evaluated at all.

Suppose we want to calculate the roots of a quadratic equation using $x = \frac{-b \pm \sqrt{(b^2 - 4ac)}}{2a}$. It would have been nice to test whether the roots are real or complex. This is how we can do it:

In [1]:
```
import scipy
# Read coefficients of quadratic equation.
a = float(raw_input("Enter a: "))
```

```
b = float(raw_input("Enter b: "))
c = float(raw_input("Enter c: "))
d = b**2 - 4.0*a*c
# Test for complex roots
if d < 0.0:
print "Complex roots"
elif d == 0.0:
x = -b/(2.0*a)
print "Equal roots at x = %f" % x
else:
x1 = (-b + scipy.sqrt(d))/(2.0*a)
x2 = (-b - scipy.sqrt(d))/(2.0*a)
print "Roots are %f and %f" % (x1,x2)
```

```
Enter a: 1
Enter b: 3
```

In [1]:
```
from IPython.core.display import Image
Image(filename='comparison.png',width = 300,height=300)
```

Out [1]:

Comparison	What it tests
a < b	a is less than b
a <= b	a is less than or equal to b
a > b	a is greater than b
a >= b	a is greater than or equal to b
a == b	a is equal to b
a != b	a is not equal to b
a < b < c	a is less than b, which is less than c

Sometimes you will want to do a more complex comparison. This is done using boolean operators such as and and or:

```
if x == 10 and y > z:
    print "Some statements which only get executed if"
    print "x is equal to 10 AND y is greater than z."
if x == 10 or y > z:
    print "Some statements which get executed if either
    print "x is equal to 10 OR y is greater than z"
```

Look at this simple script:

```
sum1 = 0.0
n = 0
while n < 10:
    x = float( raw_input("Number? "))
    sum += x
    n += 1
print "Number = %d Total = %f" % (n, sum1)
```

Here is something new: the **while** statement. The script starts off with assigning sum1 = 0.0 and n = 0, then does a loop. The loop continues so long as the test condition is true. In this case it will carry on looping as long as n < 10. In the loop, you are asked to type in a number, x, which gets added on to the previous value of sum (the statement sum +=

x is the same as sum = sum + x). In other words, the variable sum1 keeps track of the running total. At the same time, the value of n is increased by 1. This means that the loop will stop after it has run 10 times. The print statement prints the sum of all the values of x and the number of times the loop has been executed (which will always be 10). **Note** the indentation of statements within the scope of the loop. As in conditional statements, indentation is very important because it defines the limit or scope of the loop. Now look at this code:

```
sum1 = 0.0
n = 0
while 1:
    x = float( raw_input("Number? "))
    if x == 0.0:
        break
    sum1 += x
    n += 1
print "Number = %d Tota l = %f" % (n, sum1)
```

The WHILE statement is always true because any non-zero number is regarded as "true". The test comes in the statement if x == 0.0:. If a value of 0 is entered, the IF statement is true and break causes an exit from the loop. In this way you are not limited to a fixed number of items, but can signal that the list of items is finished by entering zero. Most of the time though you will need specific range for your loops, then make use of the FOR loop. e.g

```
for i in range(0,20):
    print i
```

Exercise

3. Write a script which asks for a series of numbers and prints the average. Use a loop to ask for the next number and break from the loop if the number entered is a particular number.
4. In 1897, the Indiana State House of Representatives passed a bill (House Bill 246) supposedly setting the value of π to 3. The bill was defeated in the Senate, despite support from the State Superintendent of Education. It was written by Edward J. Goodwin (who was offering his mathematical results to the State of Indiana, "free of charge"). Can you do a bit better? Write a script to calculate π using the series

$$\pi 2/12 = 1 - 1/4 + 1/9 - 1/16 + 1/25 - .... \tag{1}$$

Ask for the number of terms and print the difference between the true value of π and the value obtained from the series.

Files manipulations

So far all input has been on the keyboard (standard input) and output has been on the display (standard output). It is often more convenient to use a file for input or output or both. Here is an example where we write to a file called foo.out:

```
fp = open("foo.out","w")
a = 0
b = 1
while b < 1000:
    fp.write("%4d" % b)
    c = a+b
    a = b
    b = c
fp.write("\nThe End\n")
fp.close()
```

Let's have a close look at all this. The first statement, fp = open("foo.out","w"), opens the file (creates it if it doesn't exist or overwrites it if it does) and assigns a file pointer, fp, to it (it doesn't have to be called fp, you can call it anything). The file is used by referencing the file pointer. If you didn't want to overwrite the contents of foo.out, you could have used the statement fp = open("foo.out","a") which appends output to an existing file of the same name. In line 5, the statement fp.write("%4d" % b) writes the value of b to the file in the same way as the print statement already discussed. Note that there is no newline character in the write statement, so each number is appended on the same line. You could have used fp.write("%4d" % b) to write every number on a separate line. The fp.write("End") statement is executed at the end of the loop. It writes The End on a new line and ends with another new line. Finally, we need to close the file as soon as we are done (fp.close()). Reading from a file is equally simple. Here is a script which reads a file line by line and displays the contents on the screen:

```
name = raw_input("File name? ")
fin = open(name,"r")
for line in fin:
    print line,
fin.close()
```

So here we open the file as usual (the "r" signifies that the file is for reading) and assign the file pointer to variable fin. Next we loop through each line of the file and print the line (the variable line already includes a newline character at its end, so we need to use a comma after the print statement otherwise we get two newlines in the display). Finally, we close the file. Really simple. Let's look at some more examples. Here is a script which writes the value of x and y = exp($-$x2) to file gauss:

```
import math
f = open("gauss","w")
x = -3.0
while x < 3.0:
    y = math.exp(-x*x)
    f.write("%6.2f %10.4e\n" % (x,y))
    x += 0.1
f.close()
```

After running this script, the contents of file gauss looks like this: . . .

-2.50 1.9305e-03

-2.40 3.1511e-03

-2.30 5.0418e-03 . . .

Note the statement x += 0.1. This is the same as writing x = x + 0.1 and simply increments x by 0.1. Now let's read these data using another script:

```
f = open("gauss","r")
for line in f:
    x = float(line.split()[0])
    y = float(line.split()[1])
print x,y
f.close()
```

As usual, we open the file (for reading this time) and we start reading each line in the for loop. Remember that there are two numbers in each line. We need to split the line and read each number from the correct split portion. The split() method applied to a string (in this case the string line), splits the string into pieces if there are one or more spaces (or any whitespace) between each piece. So for example, the line

-2.50 1.9305e-03 is made up of two pieces, -2.50 and 1.9305e-03. The first piece is stored in the string line.split()[0] and the second piece in line.split()[1].

Now the whole part described above is quite outdated, while it should always work fine. A new approach which I recently discovered to read file without needing the use of loops is the loadtxt command from numpy package and the

asciitable.read command from asciitable package:

```
import astropy.io.ascii as asciitable
b = asciitable.read('read.txt')
```

That's it! Now b is an array which contain all the information from the file read.txt. If you do b[0], the output will show the 1st row of the file read.txt. If you want just the value of the 1st row and 2nd column of b then you must type b[0][1] and press enter. Asciitable is very powerful because it is very customisable and for more information consult the following link: http://cxc.harvard.edu/contrib/asciitable/

If you have 2 arrays with some data and wants to output them to a data file using asciitable you could try the following

```
import astropy.io.ascii as asciitable
import numpy as np

x = np.array([1,2,3,4,5,6,7,8])
y = np.array([2,4,6,8,10,12,14,16]) # x and y are your 2 arrays of value that you want to s

asciitable.write({'x':x,'y':y}, 'test.txt', names = ['x','y'])
```

1.1 Writing your own functions

Writing functions is a big part of any programming language and is just as important in python. I strongly recommend to write codes in a 'modular' fashion where everything is written in functions and the MAIN program only calls for all of the written functions. To define a new function you just need to use def () and always use identation after the def () so that python recognise which lines belong to the function you define and which does not. Usually you end a function with the command 'return' (just like in C). Unlike in C where functions starts with '{' and ends with '}', python only uses indentation to recognise which lines belong to functions or loops and which does not. Therefore it is of great importance for you to always use good indentation style in the code you write.

For example if you want to create a function to calculate Planck's Blackbody radiation at different wavelength, , then you could do it with the following

```
# Define Planck's function taking as inputs wavelength, wav (nm)
# and temperature T (K). The light intensity is returned. All
# units are SI. Note indentation.
import math
def planck(wav, T):
    a = 2.0*h*c**2
    b = h*c/(wav*k*T)
    intensity = a/ ( (wav**5)*(math.e**b - 1.0) )
    return(intensity)
```

Note that the use of '#' is made to comment the lines in the code – Commenting is essential if you want to become an efficient programmer whose codes can be shared around – which happens a lot in scientific collaboration.

The function has a name planck which is meaningful, though it could have been called anything. The function requires two values, the wavelength (in nm), wav, and the temperature, T, in K. It uses the fundamental constants defined globally (those constants had to be defined somewhere at the beginning of the program) to calculate the intensity. It returns the value of intensity. Indentation is essential because it is used to define the scope of the function. In the main program if you want to call the function then you could use the following:

p = planck(wav*1.0e-9,T)

(where T has already been defined earlier in the code) p has the wavelength (in nm) and the temperature (T) as inputs. The function return value is assigned to the variable p which is the intensity. In Python, functions always return a value. If you don't explicitly return a value, Python assumes the return value is "None".

An important point about functions is that they can be re-entrant. In other words, a function can call itself (recursive function). For example here is a function to calculate the factorial n!:

```
# Returns n!
def factorial(n):
    if n < 1:
        return 1
    else:
        return n*factorial(n-1)
n = int(raw_input("n? "))
print "n! = %d\n" % factorial(n)
```

If there is a need to return more than 1 specific value, list, arrays, etc ... you can just use a comma:

```
def data(strti,endi,scidata):
    x1 =[ ];y1 =[ ];yer=[ ]
    while (strti <= endi):
        x1.append(crval+(strti*cst))
        y1.append(scidata[strti])
        yer.append(scidata1[strti])
        strti = strti+1
    x1 = np.array(x1); y1 = np.array(y1)*10**15; yer = np.array(yer)*10**15
    return x1,y1,yer
```

Here strti,endi are 2 integers and scidata is an array. The returned values x1,y1,yer are all arrays Dont worry if you dont understand part of this function as we will cover array manipulation in a moment.

Exercise

In quantum mechanics the harmonic-oscillator wave function is given by $u = H_n(x)e^{-x^2}$ where $H_n(x)$ is the Hermite polynomial which obeys the recurrence relationship

$$H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x) \tag{2}$$

with $H_0(x) = 1$, $H_1(x) = 2x$. Write a recurrence function, Hermite(n,x) to calculate the Hermite polynomial and use it to compute u for n = 5 and $-5 < x < 5$ in steps of $\Delta x = 0.1$.

1.2 Lists

A list is a list of items such as:

In [3]:
```
a = ['spam', 'eggs', 100, 12.34]
print a
print a[0]
print a[3]
a[2] = "A new item"
print a
```

```
['spam', 'eggs', 100, 12.34]
spam
12.34
['spam', 'eggs', 'A new item', 12.34]
```

Note that the list can have strings,integer,float stored at the same time.

Please **Note** that a list is a set enclosed in square brackets, while a tuple is enclosed in normal brackets. The items in a tuple cannot be changed, but those in a list can be changed (as in the above example). Note that the first item in a list is indexed by the number 0. Note also that the list can mix strings with numbers. In fact, a list can contain any type of data. You can do arithmetic on item lists, for example:

In [4]:
```
a = ['spam', 'eggs', 100, 12.34]
print a[0]+a[1]
print a[2]+a[3]
```

```
spameggs
112.34
```

but for obvious reasons a[1]+a[2] (i.e. 'eggs'+100) won't work as one is a string and the other is an integer

in codes you can always define empty list by using

```
list1 = []
```

And then later inside loops or otherwise you can add data to the list by using the command append like

```
list1.append(z)
```

In the case here it will append the value of z to the list. This attribute makes list very convenient. Usually arrays are a bit more rigorously define and therefore you already need to specify the size of an array beforehand when defining it while list can increase in size during a code.

Exercise

- Given a list of numbers, {x1 , x2 , . . . , xn }, write a function which returns the minimum and maximum values. Use this function to find the minimum and maximum of 10 random numbers. You can generate a list of random numbers as follows:

```
import random            #You need the "random" module.
x = [ ]                      #Define x as a null list.
for n in range(10):              #Loop between 0 ... 9.
    x.append(random.random())        #Add a random number to the list x.
```

- Write a function which returns the average and standard deviation of a list of numbers in z. If zi is a member of z then the average, $< z >= \frac{1}{n}\sum z_i$, where n is the number of items. The standard deviation is $\sigma = \sqrt{< z^2 > - < z >^2}$ (where $< z^2 >= \frac{1}{n}\sum z_i^2$). Use the function to calculate the mean and standard deviation of z = {1, 2, 3, . . . , 10}. [The answer is $< z >= 5.5, \sigma = 2.872$]

1.3 Arrays

We have discussed lists which are items enclosed between square brackets and can be of mixed character (strings and numbers can all be part of the same list). An array is just a list which contains only pure numbers (floating point or integers). Python was not created for numerical computations and did not originally contain an array structure; this was added later. To use arrays and matrices, you need to import the NumPy package. NumPy is an extension to the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large library of high- level mathematical functions to operate on these arrays and matrices . Do note that NumPy has a structure that is written in C and therefore is faster than usual python. Because of this attribute, you have to be intelligent when optimising your codes – Usually when you write codes in C or Fortran you use lots of loops to compute, sum over things. In Python, since loops are in python itself and therefore runs slower than C, arrays are good alternative ways to boost up your computing speed. By learning to master NumPy, you can find shortcuts to optimise coding.

Let's start with arrays. Here is an example of how one can create an array and do some simple array arithmetic.

In [5]:
```
import numpy as np
x = np.array([2, 4, -11])
y = x + 0.1
print "x =",x
print "y = x + 0.1 =",y
z = 2*x
print "z = 2*x =",z
u = z + y
print "z + y =",u
```

```
x = [  2   4 -11]
y = x + 0.1 = [  2.1   4.1 -10.9]
z = 2*x = [  4   8 -22]
z + y = [  6.1  12.1 -32.9]
```

In this script above [2,4,-11] is a list and doing np.array() on it, converts it from being a list to being a numpy array. Although lists can take strings and other non-numerical types, the array() statement will not work unless all the elements in the list are numbers usually.

As an exercise, you should try the same problem using a list instead of an array. You will find that to perform array arithmetic requires the use of looping through list elements, whereas you can treat an array as if it was a vector. Using arrays in such problems is clearly more convenient than using lists. It is possible to have arrays of two or more dimensions:

In [6]:
```
import numpy as np
x = np.zeros([2,2], float)
print x
print
x[0][0] = 5.3
print x
print
x[1][0] = 4
x[0][1] = -10
x[1][1] = 16
print x
print
print x[1][0]
```

```
[[ 0.  0.]
 [ 0.  0.]]

[[ 5.3  0. ]
 [ 0.   0. ]]

[[  5.3 -10. ]
 [  4.   16. ]]

4.0
```

The zeros statement here creates a 2×2 array of floating point numbers in which all elements are zero. The result of executing this script is:

[[0. 0.] [0. 0.]]

[[5.3 0.] [0. 0.]]

[[5.3 -10.] [4. 16.]]

4.0

Arrays can be created in various ways:

- With the function array(obj), where obj is a (nested) sequence, e.g. a list [ ] or a tuple ( ).
- With the function empty(shape, dtype) which produces an uninitialized array with specified shape and typecode, e.g. empty((2,3),float).
- With the function ones(shape, dtype) producing an array initialized with ones, e.g. ones((4,5),int).
- With the function zeros(shape, dtype) producing an array initialized with zeros, e.g. zeros((3,2),float).

Here is an example which asks for the elements of a 2×3 array and writes them out:

```
import numpy as np
a = np.zeros((2,3),int)
for i in range(2):
    for j in range(3):
        a[i,j] = float(raw_input("a[%d,%d]? " % (i,j)))
for i in range(2):
    print a[i]
i = 0
j = 0
while (i < 2) and (j < 3):
    str = raw_input("i, j? ")
    i = int(str.split()[0])
    j = int(str.split()[1])
    if (i < 2) and (j < 3):
        print a[i,j]
```

Arrays are useful because you can treat a whole array like a vector. In many instances, however, you can just use a list as we have done in many previous examples. In mathematics there is really no difference between a two-dimensional array and a matrix, but in Python there is a distinct difference. Numpy contains both an array class and a matrix class. The array class is intended to be a general-purpose n-dimensional array for many kinds of numerical computing, while matrix is intended to facilitate linear algebra computations specifically. In Python you can invert a matrix, for example, but not an array.

Tricks with List/Array manipulation

Knowing how to manipulate lists/arrays is something very important in Python. The first thing that you should know is way which list and arrays are saved in the memory of the computer. Try the following:

In [7]:
```
a = [1,3,7]
```

In [8]:
```
b = a
b
```

Out [8]: [1, 3, 7]

Now lets change the value of the 2nd list element in ‘b’

In [9]:
```
b[1] = 10
b
```

```
Out [9]: [1, 10, 7]
```

```
In [10]: a
```

```
Out [10]: [1, 10, 7]
```

What did you notice? When you change ‘b’, ‘a’ automatically changes! This is because in python, when you do

a = [1,3,7]

it creates a list in its memory and then it makes ‘a’ point to that list then when you do

b = a

it does not create a new list but just points ‘b’ to the same list created earlier. Therefore any change you operate on ‘b’ will automatically change ‘a’!! If you want to create a real copy of the list ‘a’ is pointing to then you should use Slice Syntax i.e.

```
In [12]: b = a[:]
```

this means then that a new list is created in the memory and only ‘b’ now points to this new list. Using the ’:’ is called slice syntax. It means that you are using all the elements of that list in the above case. If you want only part of the list you could try

```
In [13]: c = [2,5,8,3,6]
         c[2:5]
```

```
Out [13]: [8, 3, 6]
```

As you have noticed in the example c[2:5], the slicing index is inclusive for the 1st index given (here the 1st index is ‘2’) but NON-inclusive for the 2nd index (here the 2nd index is ‘5’)

```
In [14]: c[:3]
```

```
Out [14]: [2, 5, 8]
```

```
In [15]: c[2:]
```

```
Out [15]: [8, 3, 6]
```

This method of copying is NOT VALID for arrays though say you have an array:

```
In [16]: import numpy as np
         import math as mth

         x = np.array([1,4,7,10,2]) #converting a list into an array

         y = x[:]
         y
```

```
Out [16]: array([ 1,  4,  7, 10,  2])
```

```
In [17]: y = x[:]
         y
```

Out [17]:
```
array([ 1,  4,  7, 10,  2])
```

In [18]:
```
x
```

Out [18]:
```
array([ 1,  4,  7, 10,  2])
```

For arrays if you want to copy the values of an array to another then use copy from numpy i.e.

In [19]:
```
z = np.copy(x)
z
```

Out [19]:
```
array([ 1,  4,  7, 10,  2])
```

In [20]:
```
z[1] = 17
z
```

Out [20]:
```
array([ 1, 17,  7, 10,  2])
```

In [21]:
```
x
```

Out [21]:
```
array([ 1,  4,  7, 10,  2])
```

As you can see, x has not change when you change z.

Now if you were using C and you wanted to know the mean of the numbers in array z, then you would probably have gone to write a loop to sum over all the 5 components of the array and then divide by 5 to get the mean. Lots of people still do that in python, but numpy is here to make your life easy! (In fact you might want to stop the habit of using for loops or nested for loops in python as far as possible since they run quite slowly. Instead it is better to use slicing and other little tricks that numpy offers since numpy is written in C and therefore runs very fast) Just do the following

In [22]:
```
mean_z = np.mean(z)
mean_z
```

Out [22]:
```
7.4000000000000004
```

If you want to know the standard deviation of those numbers in z then just do:

In [24]:
```
std_z = np.std(z)
std_z
```

Out [24]:
```
5.8172158288995943
```

In the same way there are lots of other built-in function inside numpy which makes your life easier and I would recomment that you check them online

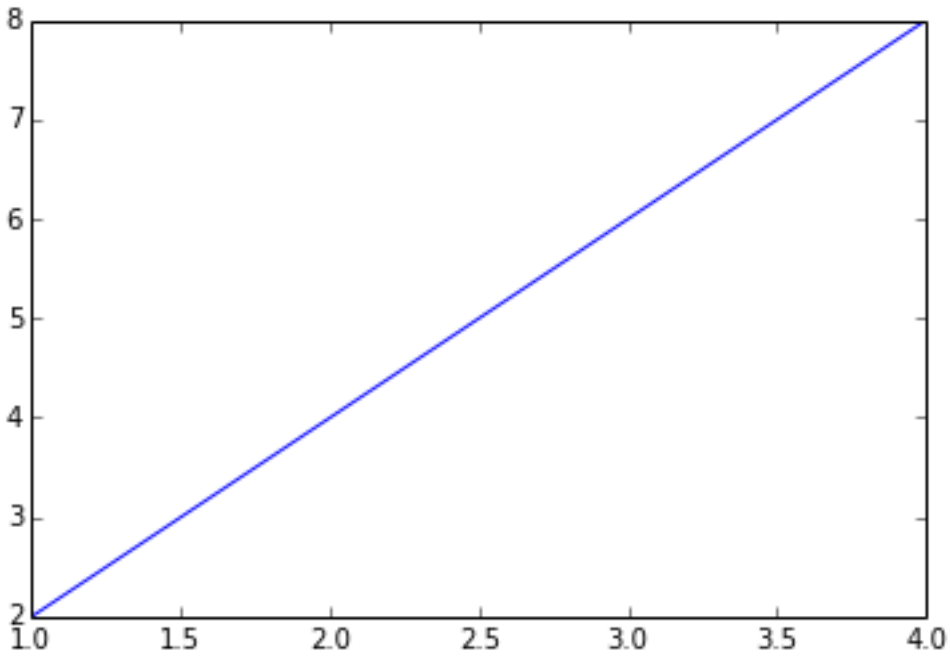
2 Quick Guide to Malplotlib

dont use the line just below in your comand line!

In [25]:
```
%pylab inline
```

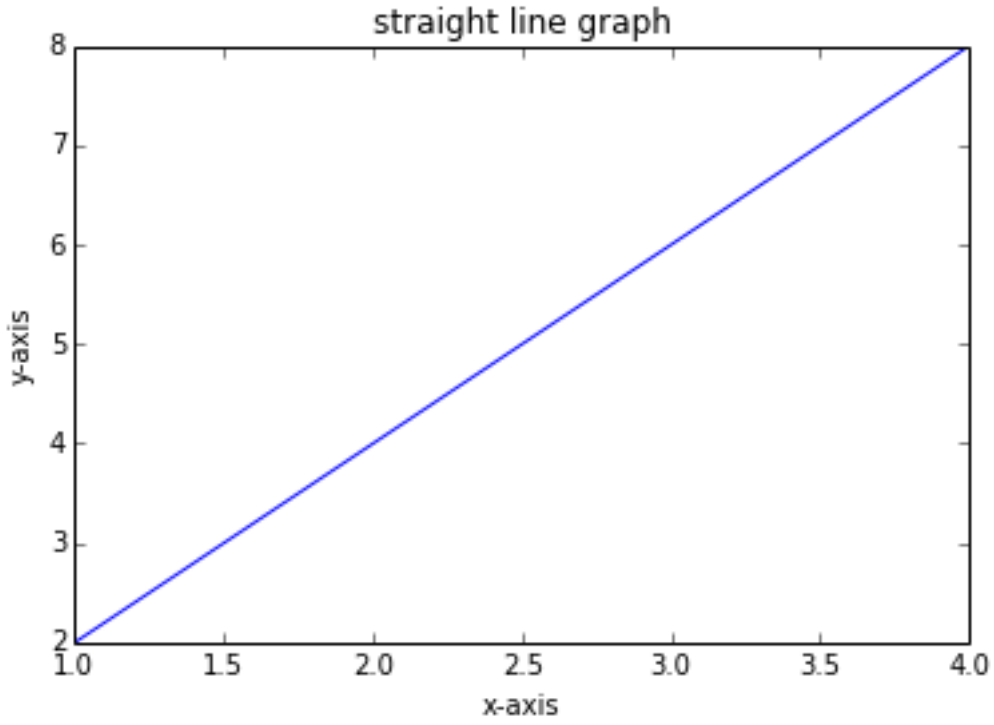
```
Populating the interactive namespace from numpy and matplotlib
```

In [27]:
```
import matplotlib.pylab as plt

x = [1,2,3,4]
y = [2,4,6,8] # saving x and y values in lists

plt.plot(x,y)
plt.show()
```



You should get a very basic plot (as shown above) in an interactive window with a straight line. Play around with the tabs in the window – you will see there are abilities to zoom, pan and save the plot. Note that when you use plt.plot(x,y) you have told Python to plot the graph but until you type plt.show() nothing will be shown on the screen. Now any good scientific plots require labelling on the axes and a title. This can be achieved via the following

In [28]:
```
plt.plot(x,y)
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title('straight line graph')
plt.show()
```



Now try the following commands to see the different results that you get

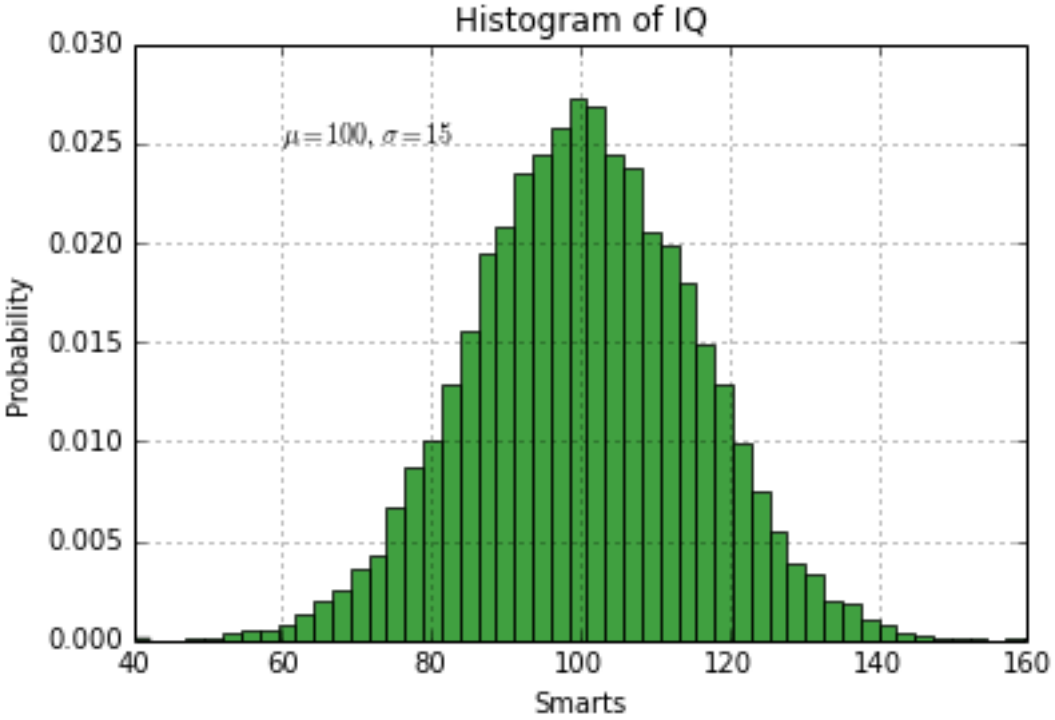
```
plt.plot(x,y, 'ro')
plt.show()

plt.plot(x,y,'b^')
plt.show()

plt.plot(x,y,'g.')
plt.show()

plt.plot(x,y,'y--')
plt.show()
```

you should get some different combination of colors and markers. Now if you want to change the linewidths: plt.plot(x,y,linewidth=2.0)

or more concisely plt.plot(x,y,lw=2.0)

Now enter the following script and see the result

In [29]:
```
import numpy as np
import matplotlib.pylab as plt
mu, sigma = 100, 15
x = mu + sigma * np.random.randn(10000)
# the histogram of the data
plt.hist(x, 50, normed=1, color='g',alpha=0.75)
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
plt.grid(True)
plt.show()
```

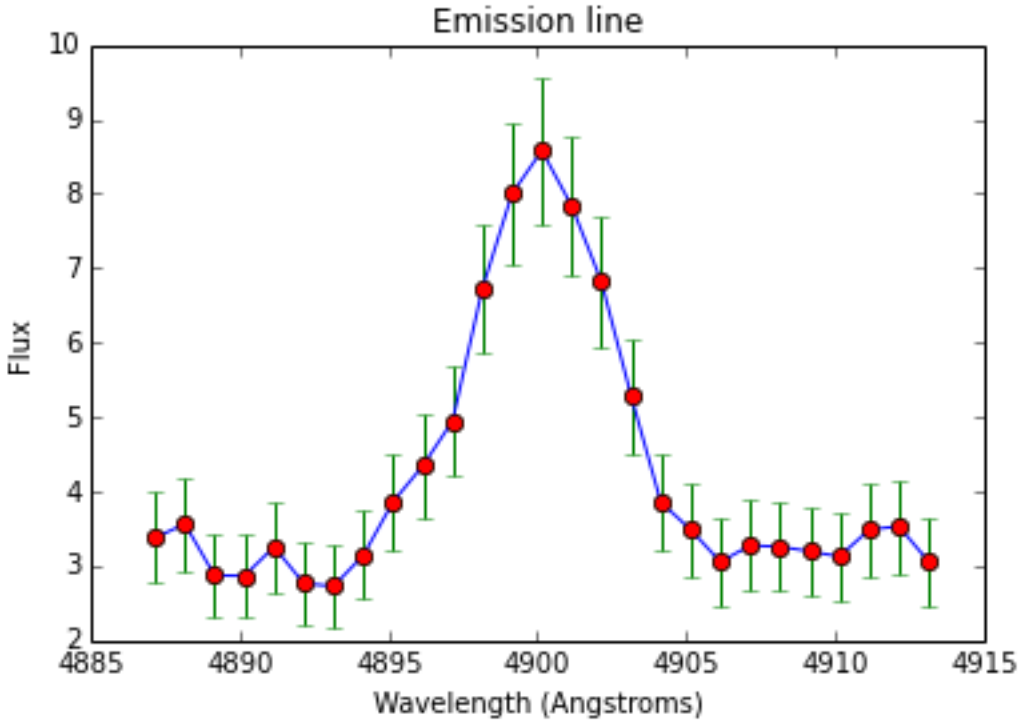


Lets analyse the script. x just creates a Gaussian distribution of random numbers centred on the mean, mu, with standard deviation sigma. To plot a histogram we use the command plt.hist and the argument 50 represent the number of bins for the histogram. Color here is just the color of the histogram while alpha is the contrast to be used when plotting (if alpha is 0 the graph appear empty and if alpha is 1 then there is a high contrast). Other interesting features are the plt.text(60, .025, r'$\mu = 100,\ \sigma = 15$') which prints the given text '$\mu = 100,\ \sigma = 15$' at the coordinates 60,0.025. Also note that you can use latex symbols i.e. to put a Greek alphabet you just need to type μ. The command plt.axis([40,160,0,0.03]) just set limits for the displayed graph. The limits in the x is from 40 to 160 and y is from 0 to 0.03

Most of the time when plotting in science you need to show your errorbars to give the reader an idea of your confidence interval of your result and you would like to do something like:

In [31]:
```
f = np.loadtxt('data.txt')
x = f[:,0] # slicing f
y = f[:,1]
yerr = f[:,2]
plt.errorbar(x,y,yerr=yerr,fmt='go')
plt.plot(x,y)
plt.plot(x,y,'ro')
```

```
plt.setp(plt.gca(), ylabel="Flux", xlabel="Wavelength (Angstroms)", title=
plt.show()
```



To learn Matplotlib best, just go on their website gallery (http://matplotlib.org/gallery.html) and just click on the plots you might find useful for you. You will be sent to a new link where you will have the plot with the code written to generate the plot below it. You can then just modify it to you liking

In []: