Siddhanth Kumar Section 102

Workshop 1: Python basics, and a little plotting

Submit this notebook to bCourses to receive a grade for this Workshop.

Please complete workshop activities in code cells in this iPython notebook. The activities titled **Practice** are purely for you to explore Python, and no particular output is expected. Some of them have some code written, and you should try to modify it in different ways to understand how it works. Although no particular output is expected at submission time, it is *highly* recommended that you read and work through the practice activities before or alongside the exercises. However, the activities titled **Exercise** have specific tasks and specific outputs expected. Include comments in your code when necessary. Enter your name in the cell at the top of the notebook. The workshop should be submitted on bCourses under the Assignments tab.

To submit the assignment, click File->Download As->Notebook (.ipynb). Then upload the completed (.ipynb) file to the corresponding bCourses assignment.

Practice: Writing Python code

The iPython Interpreter

Time to write your first python code! In Jupyter, the code is written in "Cells". Click on the "+" button above to create a new cell and type in "2+2" (without the quotes ... or with them!) and see what happens! To execute, click "Run" button or press "Shift-Enter". Also try switching the type of the cell from "Code" to "Markdown" and see what happens

In [1]:
```
2+2
```

Out[1]:
```
4
```

Practice: Peforming arithmetic in Python

If you get bored of using WolframAlpha to help with physics homework, Python can also be used as a "glorified calculator". Python code must follow certain syntax in order to run properly--it tends to be a bit more picky than Wolfram. However, once you get used to the Python language, you have the freedom to calculate pretty much anything you can think of.

To start, let's see how to perform the basic arithmetic operations. The syntax is

number operator *number*

Run the cells below and take a look at several of the different operators that you can use in Python (text after "#" are non-executable comments).

In [2]:
```
3+2 #addition
```

Out[2]:
```
5
```

In [3]:
```
3-2 #subtraction
```

Out[3]:
```
1
```

In [4]:
```
3*2 #multiplication
```

Out[4]:
```
6
```

In [4]:
```
3/2 #division
```

Out[4]:
```
1.5
```

In [5]:
```
3%2 #modulus (remainder after division) see https://en.wikipedia.org/wiki/Modulo_operati
```

Out[5]:
```
1
```

In [6]:
```
3**2 #exponentiation, note: 3^2 means something different in Python
```

Out[6]:
```
9
```

Python cares ***a lot*** about the spaces, tabs, and enters you type (this is known as whitespace in programming). Many of your errors this semester will involve improper indentation. However, in this case, you are free to put a lot of space between numbers and operators as long as you keep everything in one line.

In [7]:
```
    5     * 3   #This is valid code
```

Out[7]:
```
15
```

You are not limited to just 2 numbers and a single operator; you can put a whole bunch of operations on one line.

In [8]:
```
5 * 4 + 3 / 2
```

Out[8]:
```
21.5
```

Python follows the standard order of operations (PEMDAS) : Parentheses -> Exponentiation -> Multiplication/Division -> Addition/Subtraction. If you use parentheses, make sure every `(` has a corresponding `)`

In [9]:
```
5 * (4 + 3) / 2
```

Out[9]:
```
17.5
```

Practice: Strings vs numbers

If you're familiar with programming in other languages, you are probably aware that different *types* of things exist--you can do more than work with numbers (and not all numbers are the same type). If you'd like to work with letters, words, or sentences in Python, then you'll be using something called a string. To input a string, simply put single `' '` or double `" "` quotes around your desired phrase.

In [10]:
```
"Hello world"
```

Out[10]:
```
'Hello world'
```

Some (but not all) of the arithmetic operations also work with strings; you can add two of them together.

In [11]:
```
"Phys" + "ics"
```

Out[11]:
```
'Physics'
```

You can multiply a string by a number.

In [12]:
```
"ha"*3
```

Out[12]:
```
'hahaha'
```

This one doesn't work; try reading the error message and see if you understand what it's saying (this is a useful skill to develop).

In [13]:
```
"error"/3 #you cannot use mathematical operations between strings and integers
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [13], in <cell line: 1>()
----> 1 "error"/3

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

Practice: Printing

Up until this point, we've just been typing a single line of code in each Jupyter cell and running it. Most Python interpreters will display the result of the final thing you typed, but occassionally you want to display the results of many things in a single Python script.

In [14]:
```
"These are some numbers:"
3*2
3*3
3*4
```

Out[14]:
```
12
```

In the cell above, there are several multiplications happening but only the final result is displayed. To display everything, we simply use a "print statement" on each line.

In [15]:
```
print("These are some numbers:")
print(3*2)
print(3*3)
print(3*4)
```

```
These are some numbers:
6
9
12
```

If you'd like to print multiple things on one line, you can separate them by commas within the print statement.

In [16]:
```
print("These are some numbers:", 3*2, 3*3, 3*4)
```

```
These are some numbers: 6 9 12
```

Exercise 1: Four Fours

[Inspired by Harvey Mudd College's CS5 course] Here's an arithmetic game to try your hand at. Your task is to compute each of the numbers, from 1 through 11, using exactly four 4's and simple math operations. You're allowed to use `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), `sqrt()` (square root), `factorial()` (factorial), and `%` (modulus). You're also allowed to use `.4` (that's one 4) or `44` (that's two 4's) if you'd like. Just remember, you must use exactly four 4 digits total!

As a reminder, four factorial (denoted by $!$ in mathematics) is $4! = 4 \cdot 3 \cdot 2 \cdot 1$, and the modulus operator (usually denoted by $\mod$ in mathematics) is the remainder after division. For instance, $5 \mod 2 = 1$, $13 \mod 7 = 6$, and $14 \mod 7 = 0$.

We've given you `zero` for free, as `4 - 4 + 4 - 4`. Of course, we could have also done `44 * (.4 - .4)` or `factorial(4) - 4 * (4 + sqrt(4))`, since both of those also yield `0` (or rather, `0.0`. Why is that?) and use exactly four 4's.

In [26]:
```
### Exercise 1

from math import factorial, sqrt

print('Zero:', 4 - 4 + 4 - 4)
print('One:', (4**4)/(4**4))
print('Two:', (4/4) + (4/4))
print('Three:', 4 - (4**(4-4)))
print('Four:', 4 + ((4-4)*4))
print('Five:', 4 + (4**(4-4)))
print('Six:', 4 + ((4+4)/4))
print('Seven:', (4+4)-(4/4))
print('Eight:', (4+4)-(4-4))
print('Nine:', (4+4)+(4/4))
print('Ten:', (44-4)/4)
print('Eleven:', 44/(sqrt(4)*sqrt(4)))
```

```
Zero: 0
One: 1.0
Two: 2.0
Three: 3
Four: 4
Five: 5
Six: 6.0
Seven: 7.0
Eight: 8
Nine: 9.0
Ten: 10.0
Eleven: 11.0
```

Your final source code will be full of four fours formulas, but your final output should look like this:

```
Zero: 0
One: 1
Two: 2
Three: 3
Four: 4
Five: 5
Six: 6
Seven: 7
Eight: 8
Nine: 9
```

```
Ten: 10
Eleven: 11
```

It's ok if some of these have a trailing `.0` (`0.0` , for instance), but make sure you understand why they do!

Practice: Variables, functions, namespaces

Variables

Suppose you calculate something in Python and would like to use the result later in your program (instead of just printing it and immediately throwing it away). One big difference between a calculator and a computer language is an ability to store the values in memory, give that memory block a name, and use the value in later calculations. Such named memory block is called a *variable*. To create a variable, use an *assignment* opperator = . Once you have created the variable, you can use it in the calculations.

In [27]:
```
x = "Phys"
y = "ics!"
z = x + y     # Put 'em together
z             # See what we got!
```

Out[27]:
```
'Physics!'
```

In [28]:
```
y + x         # Backwards!
```

Out[28]:
```
'ics!Phys'
```

In [29]:
```
len(z)        # 8 characters in total ...
```

Out[29]:
```
8
```

In [30]:
```
len(z)**2     # Computing the area?
```

Out[30]:
```
64
```

In [31]:
```
z[0]          # Grab the first character
```

Out[31]:
```
'P'
```

In [32]:
```
z[1:3]        # Grab the next two characters
```

Out[32]:
```
'hy'
```

In [33]:
```
z[:4]
```

Out[33]:
```
'Phys'
```

In [34]:
```
z[:4] == x    # Test a match!
```

Out[34]:
```
True
```

In [35]:
```
z[4:] == y
```

Out[35]:
```
True
```

In [36]:
```
z[:]          # The whole string
```

Out[36]:
```
'Physics!'
```

In [37]:
```
z[::-1]       # The whole string, right to left
```

Out[37]:
```
'!scisyhP'
```

In [38]:
```
z[1::3]       # Start at the second character and take every third character from there
```

Out[38]:
```
'hi!'
```

In [40]:
```
print(z[-1])
z*3 + 5*z[-1] # Woo!
```

Out[40]:
```
!
'Physics!Physics!Physics!!!!!!'
```

Namespaces

This notebook and interpreter are a great place to test things out and mess around. Some interpreters (like Canopy) comes preloaded with a couple libraries (like numpy and matplotlib) that we will use a lot in this course. In Jupyter, you have to pre-load each package before using it. This is a good python practice anyway ! Here is an example.

In [45]:
```
import numpy as np

np.log(np.e)
```

Out[45]:
```
1.0
```

Both the function `log` and the number `e` are from the `numpy` library, which needs to be loaded into Jupyter. "pylab" adds `matplotlib` (the standard plotting tool) to `numpy`, so we will use that.

In [46]:
```
from pylab import *
log(e)
```

Out[46]:
```
1.0
```

Or type `pie([1,2,3])`, since `pie` is defined by matplotlib!

In [47]:
```
pie([1,2,3])
matplotlib.pyplot.show()  #This line is needed so matplotlib actually displays the plot
```

Note that we imported all library definitions from `pylab` into the default *namespace*, and can use the functions directly instead of having to add the name or alias of the package:

In [48]:
```
import numpy as np
np.log(np.e)
```

Out[48]:
```
1.0
```

Loading into the default namespace can be convenient, but also confusing since many names and variables are already used in ways you might not expect. When writing scripts you'll have to manually import any library you want to use. This little inconvenience is greatly worth the confusion it can save.

Functions (looking a bit ahead)

You'll often find yourself performing the same operations on several different variables. For example, we might want to convert heights from feet to meters.

In [49]:
```
burj_khalifa = 2717   #height in feet
shanghai_tower = 2073 #height in feet
```

In [50]:
```
print(burj_khalifa / 3.281)   #height in meters
print(shanghai_tower / 3.281) #height in meters
```

```
828.1011886619932
631.8195672051204
```

You could just type the same thing over and over (or copy and paste), but this becomes tedious as your operations become more complex. To simplify things, you can define a function in Python (above, you were able to use the `log()` function from the `numpy` library).

In [51]:
```
'''A function definition starts with the 'def' keyword,
followed by the function name. The input variables are then
placed in parentheses after the function name. The first line
ends with a colon'''

def feet_to_meters(height):

    #The operations to be performed by the function are now written out at the first ind
    #You can indent with tabs or a constant number of spaces; just be consistent
    converted_height = height / 3.281
    print("Your height is being converted to meters.")

    return converted_height #To return a value from a function, use the 'return' keyword
```

To use a function, simply type its name with the appropriate input variables in parentheses.

In [52]:
```
feet_to_meters(burj_khalifa)
```

Out[52]:
```
Your height is being converted to meters.
828.1011886619932
```

If you'd like a function with multiple input variables, simply separate them with commas in the function declaration.

In [53]:
```
def difference_in_meters(height1, height2):

    converted_height1 = height1 / 3.281
```

```
    converted_height2 = height2 / 3.281

    return converted_height1 - converted_height2
```

In [54]:
```
difference_in_meters(burj_khalifa, shanghai_tower)
```

Out[54]:
```
196.2816214568728
```

Practice: Formatted output

Usually the data you manipulate has finate precision. You do not know it absolutely precisely, and therefore you should not report it with an arbitrary number of digits. One of the cardinal rules of a good science paper: round off all your numbers to the precision you know them (or care about) -- and no more !

Examples:

In [1]:
```
x = 20.0 # I only know 3 digits
print(x)   # OK, let Python handle it
```

```
20.0
```

That's actually pretty good -- Python remembered stored precision ! What happens if you now use x in a calculation ?

In [4]:
```
import numpy as np

print(np.sqrt(x))
```

```
4.47213595499958
```

Do we really know the output to 10 significant digits ? No ! So let's truncate it

In [8]:
```
print('sqrt(x) = {0:5.3f}'.format(np.sqrt(x)))
```

```
sqrt(x) = 4.472
```

There are several formatting options available to you, but the basic idea is this: place `{:.#f}` wherever you'd like to insert a variable into your string (where `#` is the number of digits you'd like after the decimal point). Then type `.format()` after the string and place the variable names within the parentheses.

In [10]:
```
from math import e

print("Euler's number with 5 decimal places is {0:.5f} and with 3 decimal places is {1:.
```

```
Euler's number with 5 decimal places is 2.71828 and with 3 decimal places is 2.718
```

For more formatting options, see https://pyformat.info/

Practice

Using what you just learned, try writing program to print only 4 decimal places of π and $\log \pi$. The result should look like:

```
Hello world! Have some pie! 3.1416
And some pie from a log! 1.1447
```

In [16]:
```
from math import pi
```

```
import numpy as np

#Your print statement here

print('Hello world! Have some pie! {:1.4f}'.format(pi))
print('And some pie from a log! {:1.4f}'.format(np.log(pi)))
```

```
Hello world! Have some pie! 3.1416
And some pie from a log! 1.1447
```

Exercise 2: Coulomb force

Write a function that calculates the magnitude of the force between two charged particles. The function should take the charge of each particle (q_1 and q_2) and the distance between them, r, as input (three input variables total). The electrostatic force between two particles is given by:

$$F = k\frac{q_1 q_2}{r^2}$$

In [18]:
```
k = 8.99e9   #Coulomb constant, units: N * m**2 / C**2

def calculate_force(q1, q2, r):
    if (r == 0):
        return('Undefined Force')
    else:
        F = k * ((q1 * q2)/(r*r))
        return F

    #calculate (and return) the force between the two particles
```

Now call the function with random input values (of your choosing) and print the result with 3 decimal places. What happens if you call the function with the value $r = 0$?

In [19]:
```
calculate_force(4, 5, 888)
```

Out[19]:
```
228015.17733950168
```

In [20]:
```
calculate_force(3234234,2342342,0)
```

Out[20]:
```
'Undefined Force'
```

Practice: Simple plotting

In order to do some plotting, we'll need the tools from two commonly used Python libraries: `matplotlib` (similar to Matlab plotting) and `numpy` (NUMerical PYthon). You've seen importing at work before with `from math import sqrt`; we can also import an entire library (or a large part of it) with the following syntax:

In [21]:
```
import numpy as np
import matplotlib.pyplot as plt
```

You could have also typed `import numpy`, but programmers are lazy when it comes to typing. By including `as np`, you now only have to type the two-letter word `np` when you'd like to use functions from the library. The `np` and `plt` part of the import statements can be whatever you like--these are just the standard names.

Numpy has a lot of the same functions as the `math` library; for example we have `sqrt`, `log`, and `exp`:

In [22]:
```
np.sqrt(4)
```

Out[22]:
```
2.0
```

In [23]:
```
np.log(4)
```

Out[23]:
```
1.3862943611198906
```

In [24]:
```
np.exp(3)
```

Out[24]:
```
20.085536923187668
```

In [25]:
```
np.log(np.exp(5))
```

Out[25]:
```
5.0
```

We could have just gotten these functions from the `math` library, so why bother with `numpy`? There's another variable type in Python known as a ***list***, which is exactly like it sounds--just a list of some things (numbers, strings, more lists, etc.). We'll talk about these more at some point, but the important thing is that `numpy` has a way better alternative: the `numpy` array. Usually anything you'd want to do with a list can also be done with a `numpy` array, but faster.

Let's just demonstrate by example. Suppose we want to plot the function `x**2`. To do this, we'll plot a collection of (x,y) points and connect them with lines. If the points are spaced closely enough, the plot will look nice and smooth on the screen.

In [31]:
```
x_values = np.linspace(-5, 5, 11)
print(x_values)
```

```
[-5. -4. -3. -2. -1.  0.  1.  2.  3.  4.  5.]
```

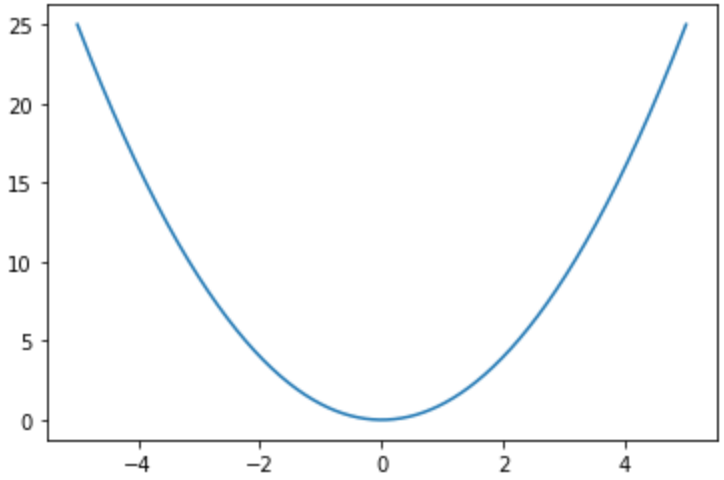
The `linspace` function from `numpy` gave us an array of 11 numbers, evenly spaced between -5 and 5. We'll want our points a bit closer, so let's change 11 to something larger.

In [34]:
```
x_values = np.linspace(-5, 5 , 1000)
y_values = x_values**2
#print(y_values)
```

To get the corresponding y values, we can just perform operations on the entire array of x values. Now, we can plot these using the `matplotlib` library.

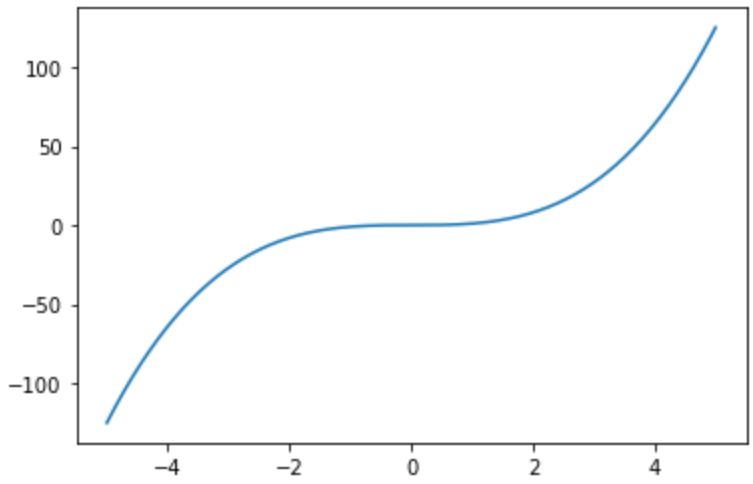
In [35]:
```
plt.plot(x_values, y_values)
```

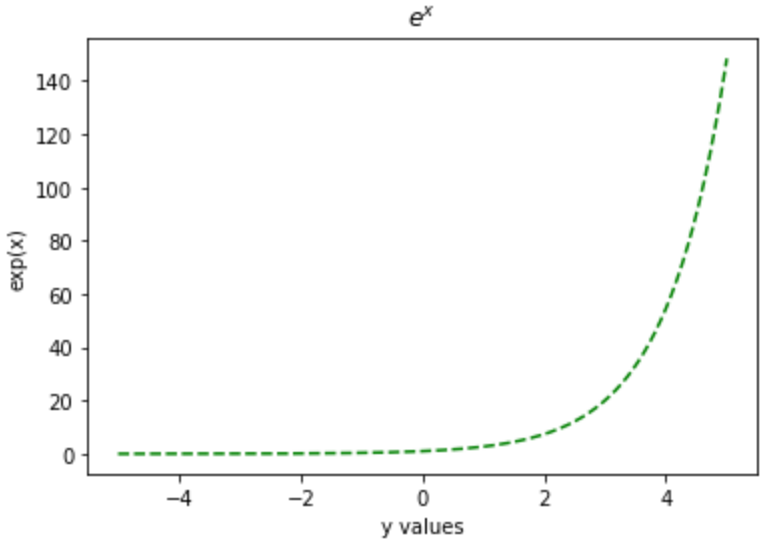
Out[35]:
```
[<matplotlib.lines.Line2D at 0x7fe6092c09a0>]
```



There's a ton of stuff you can do with `matplotlib.pyplot` or the `matplotlib` library as a whole, but here are a few basics to get you started.

In [36]:
```
plt.plot(x_values, x_values**3)  #As before, this plots the (x,y) points and connects th
plt.show()  #This forces matplotlib to display the current figure
plt.figure()  #This creates a new, empty figure
plt.plot(x_values, np.exp(x_values), 'g--')  #There are lots of optional arguments that
plt.title(r'$e^x$')          #Creates a title; you can use LaTeX formatting in matplotlib
plt.xlabel('y values')       #Label for x-axis
plt.ylabel('exp(x)')         #Label for y-axis
plt.show()
```





Exercise 3: Plotting Radioactivity Data

[Adapted from Ayars, Problem 0-2]

The file Ba137.txt contains two columns. The first is counts from a Geiger counter, the second is time in seconds. If you opened this Workshop notebook using the Interact Link (from the bCourses page), then you should already have Ba137.txt in your datahub directory.

If not, it's available here. Open the link, right-click and save as a .txt file. Then upload to datahub.berkeley.edu or move it to whichever folder you're keeping this notebook.

1. Make a useful graph of this data, with axes labels and a title.
2. If this data follows an exponential curve, then plotting the natural log of the data (or plotting the raw data on a logarithmic scale) will result in a straight line. Determine whether this is the case, and explain your conclusion with---you guessed it---an appropriate graph.

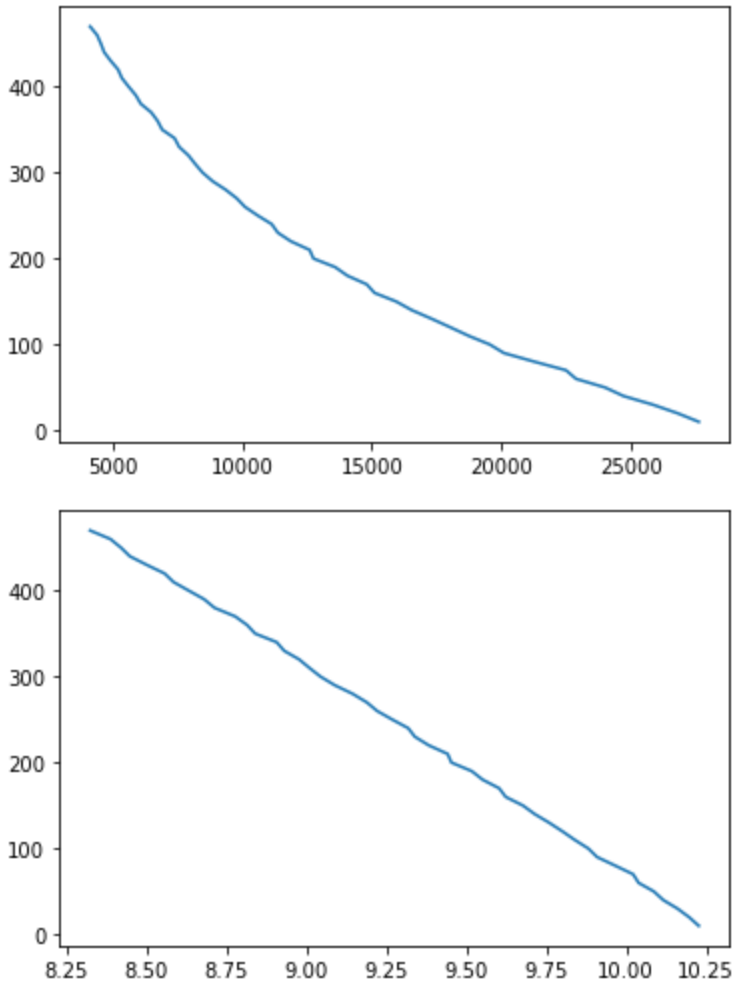
Be sure to add comments throughout your code so it's clear what each section of the code is doing! It may help to refer to the lecture notes or Ayars Chapter 0.

Try using `'x'` or `'^'` as the marker type in your `plt.plot()` functions (instead of `'g-'`, for instance), to get a single x or triangle for each data point instead of a connected line. Google if you'd like to learn more options!

Once you're through, your code should produce two graphs, one with the data, another with the natural log of the data, both labelled appropriately. It should also print out a clear answer to the question in part 2 (e.g., `Yes, the data follows an exponential curve`, or `No, the data does not follow an exponential curve`).

In [34]:
```
### Exercise 3

import numpy as np
import matplotlib.pyplot as plt

### Load the data here
counts, times = np.loadtxt('Ba137.txt', unpack = True)

plt.figure()    # Start a clean figure for your first plot
```

```
### Your code for the first plot here!
plt.plot(counts, times)

plt.figure()    # Start a clean figure for your second plot

### Your code for the second plot here!
plt.plot(np.log(counts), times) #apply to only x value, like linearization

plt.show()      # This tells python to display the plots you've made
```



Hints

Put the file in the same directory as your python file, and use numpy's `loadtxt` or `genfromtxt` function to load each column into an array for use in your plots.

If your file isn't loading correctly, it might be because your IPython working directory isn't the same as the directory your script and Ba137.txt are in.

If you'd like to learn more about what `loadtxt` does (or why the `unpack = True` option is important), type `loadtxt?` or `help(loadtxt)` into the python interpreter for documentation. Press `q` to get out of the documentation.

In [2]:
```
## Practice: Debugging

[Adapted from Langtangen, Exercise 1.16] Working with a partner, type these statements i

*Hint: Try testing the left- and right-hand sides seperately before you put them togethe

    a = 2
```

```
    a1 = b
    x = 2
    y = X + 4    # is it 6?
    5 = 5    # is it True?
    4/5 == 4.0/5.0   # is it True? (this depends on which version of Python you're using
    type(10/2) == type(10/2.)   # is it True? (again, this depends on the Python version
    from Math import factorial
    print factorial(pi)
    discount = 12%
    #ALL DEBUGGED CODE IN NEXT CELL
```

```
  Input In [2]
    [Adapted from Langtangen, Exercise 1.16] Working with a partner, type these statemen
ts into your python interpreter. Figure out why some statements fail and correct the err
ors.
              ^
SyntaxError: invalid syntax
```

In [25]:
```
a = 2
b = 2
a1 = b
x = 2
y = x + 4
print(5 == 5)
print(4/5 == 4.0/5.0)
print(type(10/2) == type(10/2.))
import math
print(math.factorial(3))
discount = 0.12
```

```
True
True
True
6
```

You're done!

Congratulations, you've finished this week's workshop! You're welcome to leave early or get started on this week's homework.