2.4.2   Graphing with Python

Programming goals

1. Getting the packages you need to get started
2. Plotting and formatting your first graph
3. Getting your data into the program
4. Adding and formatting a trendline

Getting the packages you need

Python `packages` are pre-written scripts that you can `call` with your program to help you do whatever it is you want to do. They come with pre-defined functions and values that you can use. For example, imagine you wanted the value of the constant π.

Run the cell below this one. You can run it either by clicking the `Run` button on the top (in the toolbar), or by pressing `Shift + Enter` simultaneously.

```
[ ]: pi
```

You have now seen your first Python error. Get used to errors; this one will not be your last.

You have asked Python to do something, and it does not understand what it is you have asked of it. As a result, it tells you where *it* thinks there's a problem and what the problem is (it's often right, but not always).

In this case, it says the problem is on line 1 of the cell (`----> 1 pi`) and the error is that it does not understand what you mean by `pi`.

Now compile the cell below:

```
[ ]: import math
     math.pi
```

We have `imported` the `math` package, a collection of standard mathematical values and functions. It contains, among other things, the value of `pi`, which we call by calling `math.pi`.

Some packages have horribly long names, and it would be quite cumbersome to have to type them out everytime, so we give them 'nicknames'.

See if you can understand the following code snippet:

```
[ ]: import math as m
     m.pi
```

The NumPy package

NumPy (pronounced "Num Pie", not "Num Pee") is the fundamental package for scientific computing with Python. It contains almost all scientific functions that you require, and is optimised to be significantly faster than the usual Python functions for scientific operations.

If we wish to call any functions it has, we will be using the `np` nickname. This is not essential, but it is common. Try the following:

```
[ ]: import numpy as np
     np.pi
```

The MatPlotLib package

From the Matplotlib website:

"Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code".

We will be using it to plot all our data.

You will notice a strange line underneath this: `%matplotlib inline`

This is magic, don't worry about it. It's a command that needs to be there so that figures can be printed "inline" in Jupyter notebook and be stored in the document.

The SciPy package

SciPy ("Sigh Pie") is *another* library of open-source software for mathematics, science, and engineering. We will be using it for **curve fitting**.

You can now go ahead and import all the above packages using the cell below.

```
[ ]:  import numpy as np                    # Importing the NumPy package

      import matplotlib.pyplot as plt       # Importing the Matplotlib package for plotting
                                            # "Magic" to display images inline
      %matplotlib inline

      import scipy as scp                   # Importing the SciPy package
      from scipy.optimize import curve_fit  # Importing the curve fitting module from SciPy
```

Plotting and formatting your first graph

Lists

Let's start off immediately with a simple plot. The basic syntax for creating line plots is `plt.plot(x,y)`, where x and y are lists *of the same length* that specify the (x, y) pairs that form the line.

We can define a list by simply enclosing its elements within square brackets `[]`, like so:

`listName = [element1, element2, element3]`

Exercise 1: *** In the following empty cell, define two lists, `xtest` and `ytest`, where:

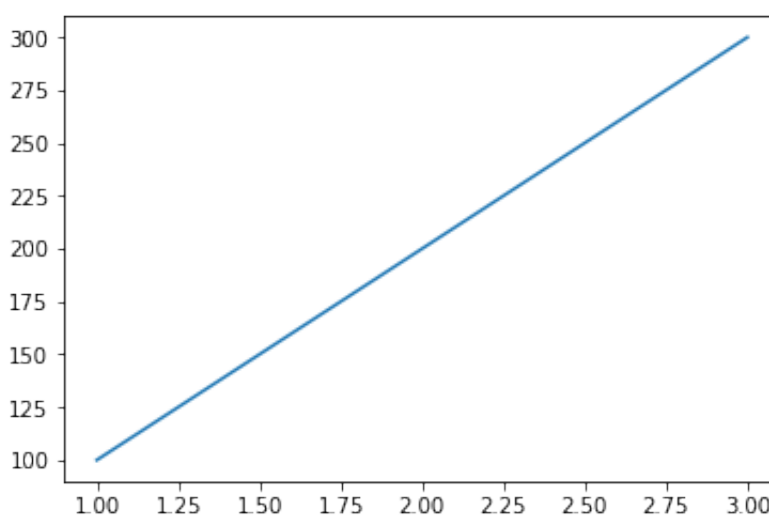`xtest` takes values 1, 2, and 3 `ytest` takes values 100, 200, and 300

```
[ ]:  # EXERCISE 1 === Fill in this cell as instructed above

      xtest = # <== Insert a three-element list of x-values
      ytest = # <== Insert a three-element list of y-values
```

Your first plot

Run the following code snippet: it will create a line plot with the above x and y values you have defined. You should get a graph like this:
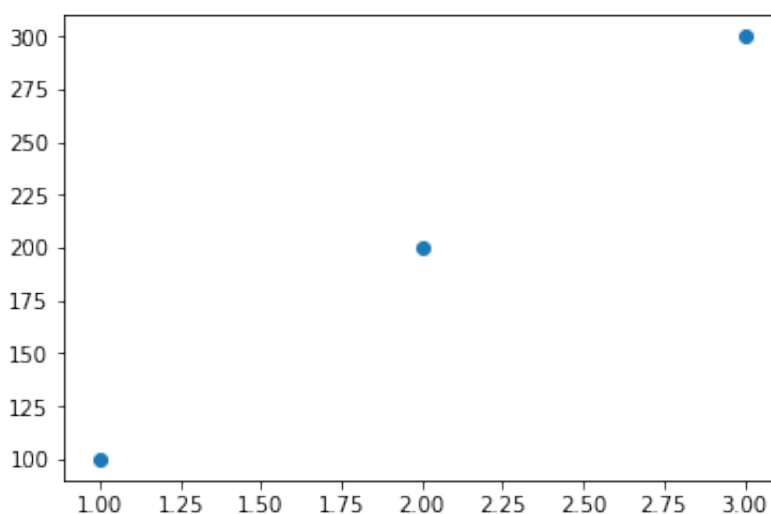
```
[ ]: plt.plot(xtest,ytest)      # Plotting a 'line' plot of ytest vs. xtest


     plt.show()                  # This function simply "shows" the graph.
                                 # it is not essential to understand how or why it works.
```

There you go, your first graph!

Of course, as mentioned in the previous section on plotting with MS Excel, **you will never plot a line from your data**. Instead, you will more often use the plt.scatter(x,y) command.

Exercise 2: *** In the following empty cell, insert one line to plot a scatter plot with the same data. After running it, you should get something like this:


```
[ ]: # EXERCISE 2 === Fill in this cell as instructed above

     #<== Insert ONE line here to plot a 'scatter' plot of ytest vs. xtest

     plt.show()         # This function simply "shows" the graph.
                        # it is not essential to understand how or why it works,
```

Formatting your graph

You will notice that it is still incomplete, you will need to label the axes. This can be done by adding modifying the plt.xlabel and plt.ylabel parameters, and re-plotting as shown below:

```
[ ]: # EXERCISE 2 === Fill in this cell as instructed above

     plt.xlabel('x values (unit)')
     plt.ylabel('y values (unit)')

     #<== Insert ONE line here to plot a 'scatter' plot of ytest vs. xtest

     plt.show()         # This function simply "shows" the graph.
                        # it is not essential to understand how or why it works.
```

You will often have to use scientific symbols in your graphs like λ and θ, and so it is helpful to learn how to do this.

Just type out plt.xlabel(r'λ (μ m)') and plt.ylabel(r'θ') in the cell below and compile it.

(The `r` present before the single quotes indicates that it is a "raw" string, and the greek letters are called using their LATEX names. You can find out more **here**.)

```
[ ]:  # EXERCISE 2 === Fill in this cell as instructed above

      # <== Insert ONE line of code to to change the x label of the plot
      # <== Insert ONE line of code to to change the y label of the plot

      # <== Insert ONE line here to plot a 'scatter' plot of ytest vs. xtest
      plt.show()      # This function simply "shows" the graph.
                      # it is not essential to understand how or why it works.
```

What if you had two sets of data to plot for the same *x* values? Try out the snippet given below:

```
[ ]:  xtest  = [1,2,3]          # A three-element list of x-values
      y1test = [100,200,300]    # A three-element list of y-values
      y2test = [50,150,250]
      plt.scatter(xtest,y1test) # Plotting a 'scatter' plot of y1 vs. x
      plt.scatter(xtest,y2test) # Plotting a 'scatter' plot of y2 vs. x
      plt.show()
```

Plotting your data

Exercise 3: *** Let's now create two simple lists of (x, y) data given in this week's handout, and plot a scatter plot of it. You should get something that looks like this:
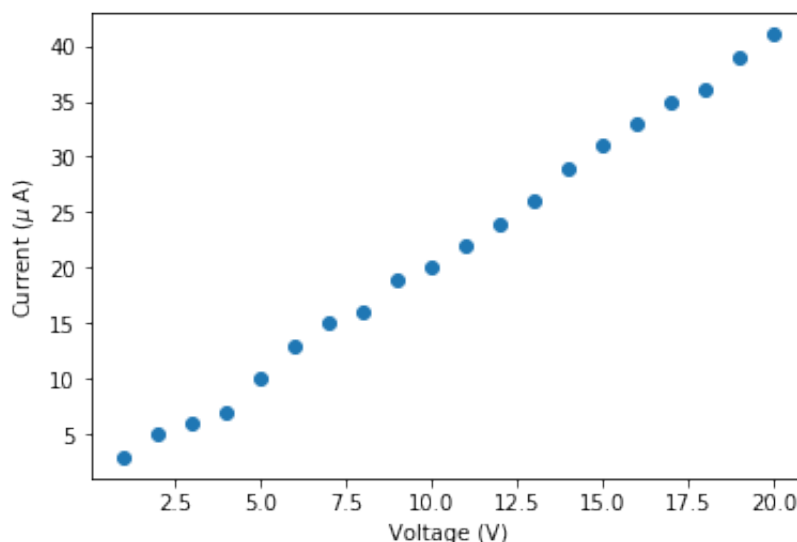

```
[ ]:  # EXERCISE 3 === Fill in this cell as instructed above

      xval = #<== Insert ONE line here with the x-axis data as a simple list
      yval = #<== Insert ONE line here with the y-axis data as a simple list

      plt.scatter(xval,yval)
      plt.xlabel(r'Voltage (V)')
      plt.ylabel(r'Current ($\mu$ A)')
      plt.show()
```

NumPy arrays

The "lists" that you've seen earlier (such as `[a,b,c]`) are those defined by default in Python. It turns out that the numpy package has its *own* version of lists, known as **arrays**. We will be using these arrays

instead of the standard lists, as they have been optimised for scientific operations.

Numpy arrays can simply be created by wrapping the earlier lists with the `np.array` command, i.e. `np.array([list])`.

Exercise 4: *** In the next cell, define `xval` and `yval` as NumPy arrays, with the same values as above. Nothing should change with your output.

```
[ ]:    # EXERCISE 4 === Fill in this cell as instructed above

        # Do the same thing as the previous cell, only this time with NumPy arrays.

        xval = #<== Insert ONE line here with the x-axis data as a NumPy array
        yval = #<== Insert ONE line here with the y-axis data as a NumPy array

        plt.scatter(xval,yval)

        plt.xlabel(r'Voltage (V)')
        plt.ylabel(r'Current ($\mu$ A)')

        plt.show()
```

Importing data

Let's say you've got your data in MS Excel, and you want to use it in Python. If you have very little data, the easiest way to do this is to type it out yourself (as we've done above). However, if you have a large amount of data, it may be better to import a `.csv` file. CSV (or "Comma Separated Value") files can very easily be created from spreadsheets like Excel (go to `File`⟶`Save As`, and select it from the drop-down list).
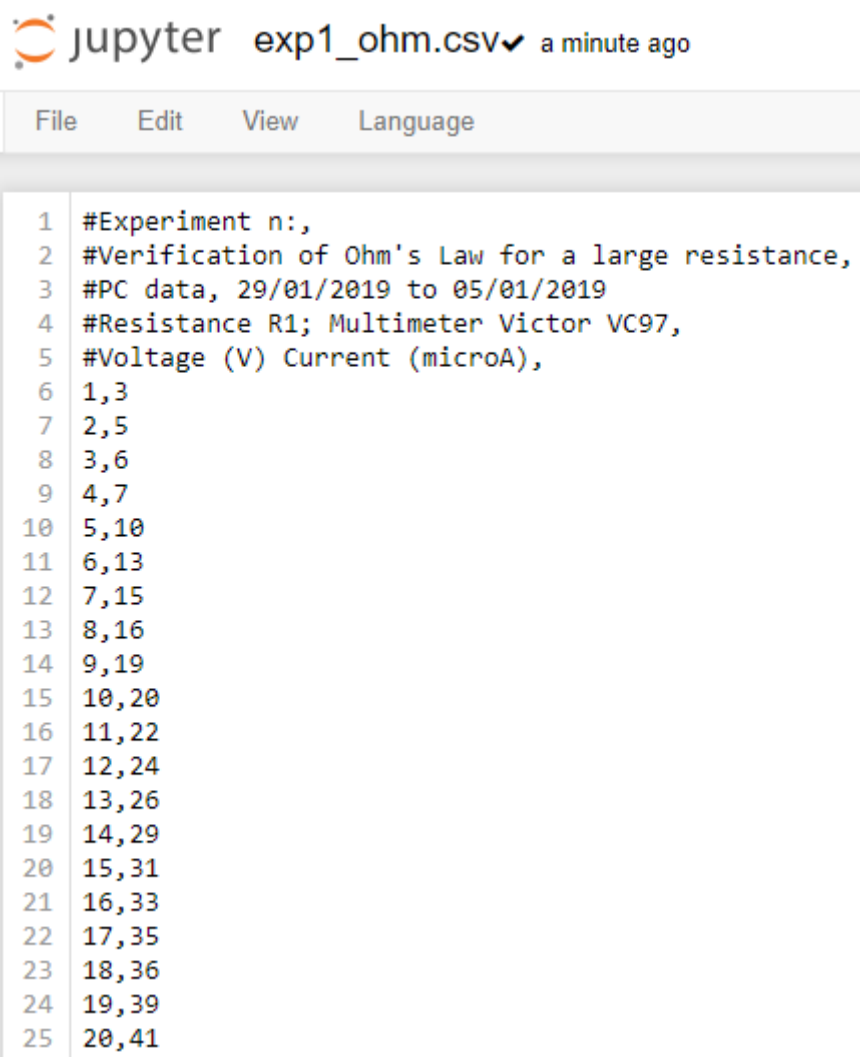
If you are asked to select a delimiter, choose `,`. This does not matter, but let us be consistent. Save the CSV file in the same folder as this Jupyter notebook, and remove all the lines that aren't the actual data.

Tip:

If creating CSV files is too difficult, here's a quick workaround:

1. On Excel, create a column with just your data values, separated by a comma. This can be done easily using the `Concatenate` function in Excel. For example: type `=CONCATENATE(A1,",",B1)` into the cell to merge them, separated by a comma.

2. Now copy the range you're interested in, and paste it in a `Something.txt` or `Something.csv` file and save it in the "data" folder that came along with this Notebook. ***

Sample data file In the "data" folder, you will see there is already a file called `exp1_ohm.csv` with sample data. You can access it in the `data` folder. In Jupyter, go to the `File` tab, and click `Open`. Navigate to the `data` folder and open `exp1_ohm.csv`. You will see something that looks like this:

```
Jupyter exp1_ohm.csv✔ a minute ago

File    Edit    View    Language

 1  #Experiment n:,
 2  #Verification of Ohm's Law for a large resistance,
 3  #PC data, 29/01/2019 to 05/01/2019
 4  #Resistance R1; Multimeter Victor VC97,
 5  #Voltage (V) Current (microA),
 6  1,3
 7  2,5
 8  3,6
 9  4,7
10  5,10
11  6,13
12  7,15
13  8,16
14  9,19
15  10,20
16  11,22
17  12,24
18  13,26
19  14,29
20  15,31
21  16,33
22  17,35
23  18,36
24  19,39
25  20,41
```

You will notice that in each row, elements of the two columns are separated by a comma, and that the first rows with the names of these columns and other details have a # symbol before them. This is very important! When you import the data into Python, this # at the beginning tells the function that's importing the data to ignore these line. (Otherwise, these lines would be a string of letters, and other lines would be numbers, and this wouldn't work). Put the # symbol in front of any lines you'd like to ignore while importing.

You are now ready to import this data using a simple `loadtxt` function that's present in the `numpy` package. We're going to learn how to do this in a slightly roundabout way:

1. Call the `loadtxt` function of the `numpy` package. You will see that it throws an error, since you have not given it the name of the file to load.

2. Now, you will start writing out the name as a string. Write out the following first `"./"` and then press the TAB button on your keyboard. You will see that Jupyter will show you the files in the current working directory (where your code is stored)! Thus, the "current working directory" is called `..`

3. You need to get to the `data` folder which is one folder up. The standard way to get to the folder above it by using `../`. Type this out, and press the TAB button.

4. Select the `data` folder, and then press the TAB button again. Now select `exp1_ohm.csv`.

5. Running this will *still* cause an error, since you haven't told Python that the different columns are separated by `,`s. Add a comma after the string, and type out `delimiter = ","`. This tells Python that the data in each row is separated ("delimited") by commas.

Tip:

Using the TAB key to autocomplete commands is a fantastic trick, useful not only here but also on most UNIX-type machine terminals (like Macs). ***

Exercise 5: ***

Complete the steps detailed above and run them in the empty cell below, you should get an output that looks like this:

```
array([[ 1.,   3.],        [ 2.,   5.],        [ 3.,   6.],        [ 4.,   7.],        [
5., 10.],        [ 6., 13.],        [ 7., 15.],        [ 8., 16.],        [ 9., 19.],
[10., 20.],        [11., 22.],        [12., 24.],        [13., 26.],        [14.,
29.],        [15., 31.],        [16., 33.],        [17., 35.],        [18., 36.],
[19., 39.],        [20., 41.]])
```

This is a two-dimensional array (very much like a matrix). It is useful to learn how to manipulate them, and I've added a short optional section about them at the bottom. But for now, we won't really use them since there's a much simpler way to "unpack" your data directly in a format you can use.

```
[ ]: # EXERCISE 5 === Fill in this cell as instructed above


     #<== Insert ONE line here to load the file exp1_ohm.csv from the data folder
```

Unpacking data   Wouldn't it be nice if Python understood that the first column was one variable's data and the second another variable's data? It turns out that the NumPy package allows for just this, using another option called unpack.

Using this option, you can equate **any number** of variables to this data, and Python will automatically send the first column to the first variable, the second to the second, and so on. Thus, if you did:

```
x1,x2,x3,...,xn = np.loadtxt(...,unpack=True)
```

to a file that contained n columns, the first column would be stored in x1, the second in x2 and so on!

Exercise 6: *** In the cell below, write down the code to take the first column as xpoints, and the second column as ypoints.

```
[ ]: # EXERCISE 6 === Fill in this cell as instructed above


     #<== Insert ONE line here to load the file exp1_ohm.csv from the data folder,
     # and unpacking by column int two variables `xpoints` and `ypoints`
```

Adding a Trendline

Adding a trendline in Python is not as obvious for the simple linear graphs as it was in Microsoft Excel, but provides much more functionality. The idea we will use is the following:

1. We will define a function (say) f which accepts some parameters (x,a,b,c...) (how many depends on what type of function we're fitting) and returns a value.

2. We will then call scipy.optimize's curve_fit function with f and our data (xpoints and ypoints) which will automatically vary the parameters to give us the best values for a,b,c...

This is shown in the following snippet:

```
[ ]: def f(x, a, b):          # Define a function `f` which `returns` a value of a*x+b
         return a*x + b       # This line makes sure that the function returns the above␙
     ↪value


     par, covariance = curve_fit(f, xpoints, ypoints)


     print("Variable par is this array:",par)
```

```
print("")
print("The slope is:",par[0], " and the intercept is: ",par[1])

m = np.round(par[0],3)   # We use the `numpy.round` function to round to 3 decimal␣
 ↪places
c = np.round(par[1],3)
```

The line `par, cov = curve_fit(f, xpoints, ypoints)` might require some explanation: the `curve_fit` function returns **two values** by default, an array with the *parameters*, which we have called `par` here, and an array with the *covariance* which is a statistical concept that you do not need to interest yourself with now.

All you need to know is that `par[0]` is the value of `a` (the slope of your line) and `par[1]` is the value of `b` (its intercept).

Let's use this now to create an array of "theoretical" y values, using the formula $y(x) = mx + c$ — a straight line.

```
[ ]: ytrend = m*xpoints+c      # Creates an array of y values corresponding to the␣
      ↪xpoints,
                              # which satisfy the trendline given by the parameters in␣
      ↪[par]

     plt.plot(xpoints, ytrend, '--',color="red") # Plot a red dashed line of ytrend vs.␣
      ↪xpoints
     plt.xlabel(r'Voltage (V)')
     plt.ylabel(r'Current ($\mu$ A)')
     plt.show()
```

Now, we can simply *combine* the above two graphs, as shown below.

You can also add an equation using the `plt.text` command, which allows you to add a string of text (here, the variable `eqn`) at specified coordinates (in our case, $x = 4$, $y = 14$), with font size 12pt. If you're plotting a straight line, you don't need to change `eqn`, it takes the value of the slope and intercept defined earlier, converts them into strings and adds them to the graph. (Placing `{c:+}` simply states that if `c` is a positive number, it is made into a string `"+c"` instead of just `"c"`(see **here** for more).

```
[ ]: plt.scatter(xpoints,ypoints)                        # Plotting the data-points
     plt.plot(xpoints, ytrend, '--',color="red")          # Plotting the trend-line

     slope_string = round(m,3)                            # Rounding off the slope to 3␣
      ↪digits
     intercept_string= round(c,3)                         # Rounding off the intercept to 3␣
      ↪digits

     eqn = 'y(x) = '+f'{slope_string}'+'x'+f'{intercept_string:+}'  # The equation string

     plt.text(4, 14,eqn,fontsize=12)                      # Displaying the above string


     plt.xlabel(r'Voltage (V)')                           # Formatting the axes
     plt.ylabel(r'Current ($\mu$ A)')
     plt.show();
```

FIN

Slightly more advanced topics

Two-dimensional arrays *(optional)* Sometimes you might want to include your data as an array of (x, y) points, for example something that looks like:

```
z = [[ 1.   3.],[ 2.   5.],[ 3.   6.],[ 4.   7.],[ 5. 10.],[ 6. 13.],[ 7. 15.],[ 8. 16.],[
9. 19.],[10. 20.],[11. 22.],[12. 24.],[13. 26.],[14. 29.],[15. 31.],[16. 33.],[17.
35.],[18. 36.],[19. 39.],[20. 41.]]
```

In this case, to plot it, you would first have to take all the xs, and then all the ys. We do this using the highly useful : symbol as follows:

```
[ ]: z = np.loadtxt("../data/exp1_ohm.csv",delimiter=",")


print("The 2D array z:")
print(z)


xpoints = z[:,0]


print()
print("The slice of x values:")
print(xpoints)
```

The : symbol here acts like a **wildcard**. Suppose our array z looks as follows:

```
[[ 1.   3.]  [ 2.   5.]  [ 3.   6.]  [ 4.   7.]  [ 5. 10.]  [ 6. 13.]  [ 7. 15.]  [ 8.
16.]  [ 9. 19.]  [10. 20.]  [11. 22.]  [12. 24.]  [13. 26.]  [14. 29.]  [15. 31.]
[16. 33.]  [17. 35.]  [18. 36.]  [19. 39.]  [20. 41.]]
```

The : (called the *slicing* operator) gives us a 'slice' of the data: **all values in the column on the left**. (This is Column "0". Column "1" would be the one on the right.)

Exercise 7: ***

Complete the following cell: Write down a line of code to collect the y points and create an array ypoints.

```
[ ]: # EXERCISE 7 === Fill in this cell as instructed above


#<== Insert ONE line of code here to get the data as a 2D array z


xpoints = z[:,0]
#<== Insert ONE line of code here to get the ypoints data
```

For those interested in more advanced fits *(optional)* In order to plot a polynomial trendline, we will evaluate this (polynomial) function at the different xpoints and plot a dotted line over the earlier graph.

In order to do this, we will be using the numpy.polyval function which accepts two variables – an array of coefficients ($[a_n, a_{n-1}, ..., a_2, a_1, a_0]$) and a value of x – and prints out $y(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_2 x^2 + a_1 x + a_0$.

Thus, calling np.polyval([a,b],x) would give you $y(x) = ax + b$. If we used an *array* of x points, we'd get an *array* of y values!

We'll try to plot the line $y(x) = -x + 4$ to see how it works. The parameters will be [-1,4]. The resulting array will contain the values of y at these points.

```
[ ]: xvalues = [0,1,2,3,4,5]                  # A sample array of x values
     yvalues = np.polyval([-1,4],xvalues) # y values corresponding to -1*x + 4


     print(yvalues)                          # Printing out the yvalue array
```

```
plt.plot(xvalues,yvalues,'--')        # Plotting yvalues vs. xvalues in a line plot,
                                      # with a dashed line (given by '--')
```

Using np.genfromtxt *(very optional)*   Apart from the simple `loadtxt` function, `NumPy` also has a slightly more powerful `genfromtxt` function which allows you to deal with `.csv` files that have missing values and so on. However, at your level, I think there is no difference at all between the two, and `loadtxt` is easier to remember!

You can use the `unpack` function here as well, but for illustrative purposes, I have used the slicing operator `:`.

```
[ ]:  data = np.genfromtxt("../data/exp1_ohm.csv", delimiter=",") # Looks in the current␣
      →directory
                                                      # for the csv (or txt) file and
                                                      # imports it, with commas being
                                                      # treated as delimiters.

      xpoints = data[:,0]                             # Get the first column of data,
                                                      # saving it to xpoints

      ypoints = data[:,1]                             # Idem for second column and␣
      →ypoints.

      plt.scatter(xpoints,ypoints)
      plt.xlabel(r'Voltage (V)')
      plt.ylabel(r'Current ($\mu$ A)')
      plt.show()
```