Introduction to computing and data handling 2023

```
>>>plt.plot(lamda,Weins)
#2nd case
>>>plt.plot(lamda, approx)
```

Why do we have to make variable or function named *approx*? What goes wrong in plotting if you have a 30 element array for RJ and a 70 element array named Weins?

3rd class *numpy*, plotting continuing. 1st Sep

Need of log scale - if there is a dramatic variation along any of the axes, use log scale to capture the behaviour of the quantity better. There are two ways a log plot can be done.
Either using functions *plt.loglog*, *plt.semilogx*, & *plt.semilogy*
OR
Using functions *plt.xscale* & *plt.yscale*

Next our aim is to generate (nearly) "publication quality" figures.

Find out (i) how to label axes using functions *plt.xlabel* and *plt.ylabel*, (ii) how to set x and y ranges using *plt.xlim* and *plt.ylim*, (iii) how to add a legend to the curve using *label="text"* in the plot command along with *plt.legend* afterwards, (iv) how to change plot styles.

There are more controls possible. You can explore yourself.

Back to numpy array

Numpy arrays are powerful tools for numerical computing[*]. So far we have only seen 1-dimensional arrays. It is possible to create arrays of any dimensions.

Let us create

```
>>>y = np.array([ [2.,  15.,  -5.,  0.9])
>>>z = np.array( [    [2.,  15.,  -5.,  0.9],[10.,  25.,  8.,  0.001]    ]  )
```

Use functions

len

np.shape

np.size

and see how the results differ. Which function will you use to find out the dimensions of an n-dimension array ?

Create an n dimension array of arbitrary n using *np.reshape*

```
>>>y.reshape(2,2)
```

```
>>>w = np.arange(1,5)
>>>w.reshape(2,2)
```

Follow the same method, use arange and create a 3 dimensional array.

[*] Numpy arrays can be made out of strings also. But we'll focus on floats and integers.

Array slicing.

To slice a numpy array, the syntax is *a[start:end:step, start:end:step, …..]*
The slicing details on each axis is given separated by comma. The start:end:step will give the details of how an axis should be sliced. That is, *[start:end:step]* will result in index = *start, start+step, start+2.*step…. start+n*step,* where *start+n*step< end*

Let us take it as a write it in this form. The below is a 3X4 matrix.

a_{00}	a_{01}	a_{02}	a_{03}	**Zeroth row**
a_{10}	a_{11}	a_{12}	a_{13}	**First row**
a_{20}	a_{21}	a_{22}	a_{23}	**Second row**
Column 0	**Column 1**	**Column 2**	**Column 3**	

Let us say, row number is decided by index j and column number by index k.
Operation [0:1:1] on row will give row number 0, ie, j=0, operation [0:1:1] on column will give column number 0, ie., k=0 leading to a_{00} as the answer. If you consider [0:3:2] on the row, the row indices resulted are j=0,2. If you consider [1:3:1] on the column, it will lead to k=1,2 (3 is not included that is, the end index is not included). Therefore the final result will be

a_{01} a_{02}

a_{21} a_{22}

Example: Use the numpy array *slice_test* given below and make various slices.
This matrix has three rows and 4 columns
>>>*np.shape(slice_test)* #will give you(3,4)

2.0	15.0	-5.0	0.9	**Zeroth row**
10	25	8	0.001	**First row**
12.0	89	0.5	-2.4	**Second row**
Column 0	**Column 1**	**Column 2**	**Column 3**	

Try out different slices and see what you get. On the other way round, see what operation you need to get a particular slice.

For example the above slicing will lead to a 2X2 matrix

15.0 -5.0

89. 0.5

If you do z [0:1:1,0:1:1] you will get the zeroth row and zeroth column, i.e, just element 2.