3. Intro to Numpy

"Simulation is the new experimentation." - Unknown

NumPy

Ordinary Python can do quite a bit “out of the box”. The list data structure is very powerful, and much more flexible than a C or Fortran array. Also, the built-in math libraries contain all the common mathematical functions, even for complex numbers (cmath). Finally, we can read in the contents of a data file with a for-loop. So what more could we want?
The answer is that NumPy is built to make it easier to handle numerical data and do calculations for scientific applications. The core of NumPy is the NumPy array. This behaves a little like a list, but adds a huge amount of functionality which wouldn’t make sense in the general-purpose list structure, but which is invaluable if you’re working with large amounts of numerical data.
Coupled with this, NumPy allows efficient matrix arithmetic and includes a lot of efficiently implemented linear algebra and matrix manipulation functions to use with the numpy array – inverting matrices, finding eigen values or just transposing a matrix. It even includes some powerful one-line functions for reading and writing data from or to formatted text files that spreadsheets like, or even some binary formats.

The Basics Of NumPy

To get access to all of this, we import `numpy` as for any python module:

```
import numpy as np
```

You’ve already used the `range()` function a lot, and its very useful, but it only works with integers. However, in `numpy` there’s a function called `arrange()` (array range) which can take arbitrary floating-point increments:

```
x = np.arange(-0.1,0.5,0.007)
print(x)
```

Output:

```
array([-0.1  , -0.093, -0.086, -0.079, -0.072, -0.065, -0.058, -0.051,
       -0.044, -0.037, -0.03 , -0.023, -0.016, -0.009, -0.002,  0.005,
        0.012,  0.019,  0.026,  0.033,  0.04 ,  0.047,  0.054,  0.061,
        0.068,  0.075,  0.082,  0.089,  0.096,  0.103,  0.11 ,  0.117,
        0.124,  0.131,  0.138,  0.145,  0.152,  0.159,  0.166,  0.173,])
```

Did you notice the `np.arange()` format? Remember in the previous lesson we spoke about the term `method` is like a `function` and has a dot before it? You can see all `functions` we will be referring to here are actually better known as `methods` as they form part of a `library`. To keep things simple we will keep calling them functions.

There is a similar function called `linspace()`, which instead of using a step size as a parameter, instead lets you specify how many steps you want to use:

```
x = np.linspace(0,10.0,30)
print(x)
```

Output:

```
array([  0.        ,   0.34482759,   0.68965517,   1.03448276,
         1.37931034,   1.72413793,   2.06896552,   2.4137931 ,
         2.75862069,   3.10344828,   3.44827586,   3.79310345,
         4.13793103,   4.48275862,   4.82758621,   5.17241379,])
```

Note that this includes both the starting-point (0) and the endpoint (100), unlike `range` and `arange`.

If you're a Matlab or Octave user, this may seem familiar to you. That's deliberate, as much of `numpy` is written with the intention of making a transition from Matlab as painless as possible.

So `np.eye`, `np.ones` and `np.zeros` will behave mostly as you expect (if you don't know Matlab, a quick perusal of the docs will tell you what these do, if you can't guess from the names).

The good news is that `numpy` arrays are mostly compatible with lists, and conversion between the two is easy:

```
a = [1,2,3]
aa=np.array(a)
```

```
b = [[1,2,3],[4,5,6],[7,8,-1]]
bb=np.array(b)
print(bb)
```

Output:

```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8, -1]])
```

Now, if you want to find the inverse matrix of bb, first check if its possible:

```
np.linalg.det(bb)
```

This tells us that the determinant is a positive number – so we can invert it

```
cc = np.linalg.inv(bb)
print(cc)
```

Output

```
array([[-1.76666667,  0.86666667, -0.1       ],
       [ 1.53333333, -0.73333333,  0.2       ],
       [-0.1       ,  0.2       , -0.1       ]])
```

Notice that the `det` (determinant) and `inv` (inverse matrix) are part of `numpy.linalg`, a sub-module within `numpy` dealing with advanced linear algebra functions.

Some of the more simple matrix operations like matrix multiplication (dot product) are in base `numpy`, for instance using the array bb above:

```
dd = np.dot(bb,np.linalg.inv(bb))
print(dd)
```

Output:

```
array([[  1.00000000e+00,   1.11022302e-16,  -5.55111512e-17],
       [ -6.66133815e-16,   1.00000000e+00,  -1.11022302e-16],
       [  4.16333634e-16,   1.38777878e-16,   1.00000000e+00]])
```

Which will give you an identity matrix, with ones on the diagonal and zeros (or very small numbers – computers make rounding errors) everywhere else.

A shorthand for doing matrix multiplication is the `@` symbol:

```
bb@np.linalg.inv(bb)
```

Be careful, though! `numpy.invert` doesn't do the same thing as `numpy.linalg.inv`. as it the former does a bitwise inversion, flipping ones to zeros and vice versa in each number in the array.

NumPy Arithmetic

`Numy` arrays can be added together or subtracted just as you'd expect:

```
p=np.array([1,2,3])
q=np.array([-3,-4,-5])
r=p+q
s=p-q
print(r)
print(s)
```

Output:

```
array([-2, -2, -2])
array([4, 6, 8])
```

For this to work, however, all arrays have to be the same size and shape.

You can check the size (number of elements) in a `numpy` array like this:

```
print(p.size)
print(p.shape)
print(p.ndim)
```

Output:

```
3
(3,)
1
```

The `shape` (number of rows and columns) is a `list` or `tuple` containing the number of elements along each axis of the array (eg. the number of rows and columns in a two-dimensional array) and number of dimensions `ndim` (1D is a row, 2D has rows and columns, 3D has rows, columns

and depth, and so on).

Note that `numpy` sometimes distinguishes between a 1-dimensional array and a 2-dimensional array with only one column or row. This can cause problems but can easily be fixed by manipulating the array shape.

The `numpy` function `reshape` allows you to rearrange the array so that the elements are arranged in however many rows and columns as you want, as long as the rows multiplied by the columns stays equal to the total number of elements in the array.

So, to convert p from a 1D to a 2D array, we do this:

```
p2d=np.reshape(p,[1,3])
print(p2d)
```

Output:

```
array([[1, 2, 3]])
```

The `list` parameter after `p` is a `list` that tells `reshape` how many rows and columns we want. In this case there is only one row, but three columns. Use `ndim` and `shape` to confirm the shape of `p2d`.

A last point about `numpy` array shapes: a 2D array can be transposed (rows and columns flipped) by putting a `.T` after the variable name, for instance the array p2d in the example would be transposed as `p2d.T`

Returning to the subject of `numpy` arithmetic, `*` and `/` do elementwise multiplication and division (again, only if the shape of the matrices are appropriate – they must have the same number of columns at least).

For example:

```
f=np.array([1,2,3])print(p2d)
g=np.array([2,2,-3])
f*g
```

Output:

```
array([ 2,  4, -9])
```

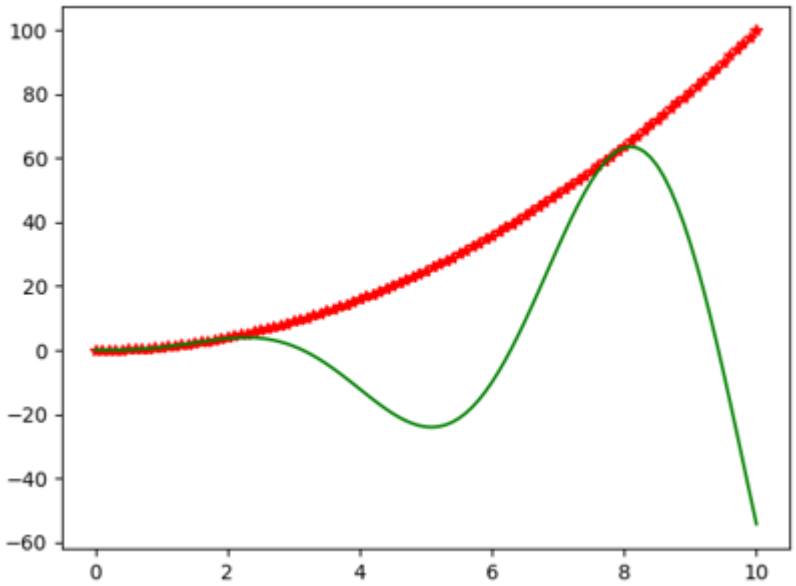
Compare this with the non-mathematical way in which `+`, `-` and `*` operate on python lists.

Make sure you don't get confused between the two!

NumPy Unique Features

`Numpy` has its own maths functions (`sin`, `cos`, `tan`, `sqrt` and so on) that operate on each element of an array. Here's a quick demonstration script on two different plots just to show you how powerful `numpy` is:

```
import numpy as np
x = np.arange(0,10.1,0.1)
y1=x*x
y2=x**2*np.sin(x) # note that ** raises each element to a power, in this case 2.
#just for fun, lets plot y1 and y2 relative to x using matplotlib (we'll do more on that later)
import matplotlib.pyplot as plt
plt.plot(x,y1,"r*")
plt.plot(x,y2,"g")
plt.show()
```



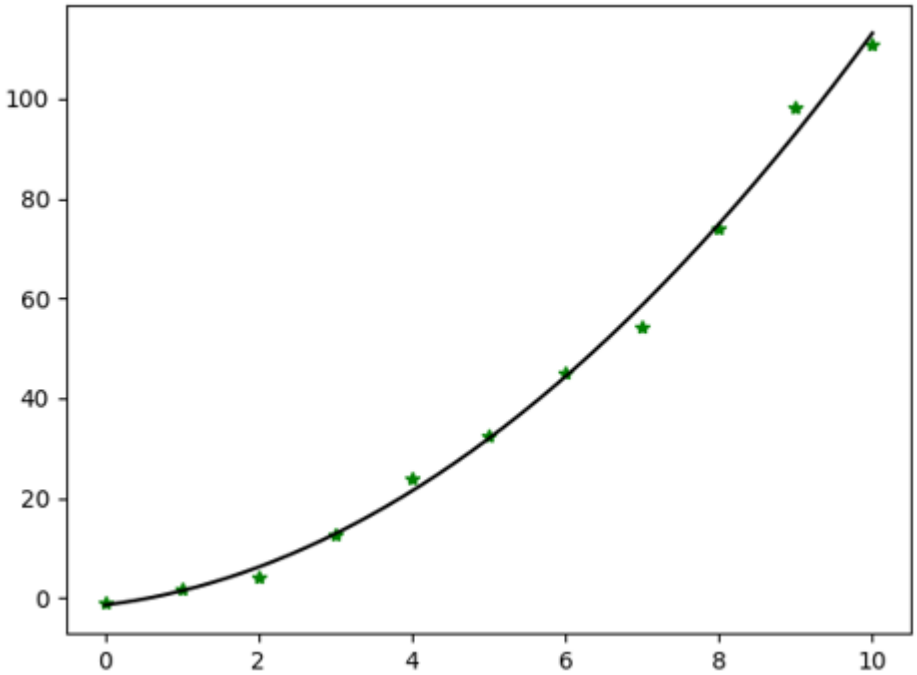
In python, you'd either have had to use a for-loop or the `map()` function to do the same thing we did with `numpy's` `np.arange` and `np.sin` methods. Either way would have required a few more lines.

If you want to do a polynomial curve-fit to x-y data, you use `polyfit`.

You can then generate a data series to plot or otherwise use by means of `polyval`.

```
import numpy as np
x = np.arange(0,11)
y=x**2+3*x*np.random.rand(x.size)-1
p = np.polyfitx,y,2)
xfit=np.arange(0,10.01,0.01)
yfit=np.polyval(p,xfit)
import matplotlib.pyplot as plt
plt.plot(x,y,"g*")
```

```
plt.plot(xfit,yfit,"k")
plt.show()
```



Note that for this example, we used the function `rand` from the `numpy` submodule random to introduce some scattter to a polynomial. Note that `rand` generates a `numpy` array of random numbers. There are other functions such as `ranf()` or `randint()` which generate floating point numbers or integers respectively, and can be given upper and lower bounds, etc. `uniform()` will approximate a uniform random distribution.

`Numpy` also has functions to calculate the `sum`, the `mean` and other statistical quantities for an arbitrary array.

One of the optional parameters to the `mean` and `sum` functions is `axis`. This allows you to average each column (axis=0) or row (axis=1) and so on for higher dimensional `numpy` arrays.

```
a=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(np.mean(a))
print(np.mean(a,axis=0))
print(np.mean(a,axis=1))
```

Output

```
5.0
array([ 4.,  5.,  6.])
array([ 2.,  5.,  8.])
```

There are many more interesting and useful tricks that you can do with `numpy` arrays; here is one that is very powerful but not obvious, which is comparing values elementwise:

```
k=np.array([4,3,5])
m=np.array([2,3,6])
print(k>m)
print(k<=m)
print(k==m)
```

Output:

```
array([ True, False, False], dtype=bool)
array([False,  True,  True], dtype=bool)
array([False,  True, False], dtype=bool)
```

You will see that the above returns an array of `boolean` values, which is the same size as `k` and `m`.

Each element from the two arrays is compared according to the inequality (or equality) operator, and a `True` or `False` value is put in the corresponding place in the `boolean` array.

This can be very useful, especially when you recall that in `python`, a `boolean` value of `True` is really just 1, and `False` is 0.

Also, if you use an array of `booleans` as the `index` to an array of identical shape, it will return only those values that are at positions corresponding with a value of `True` in the `index` `boolean` array.

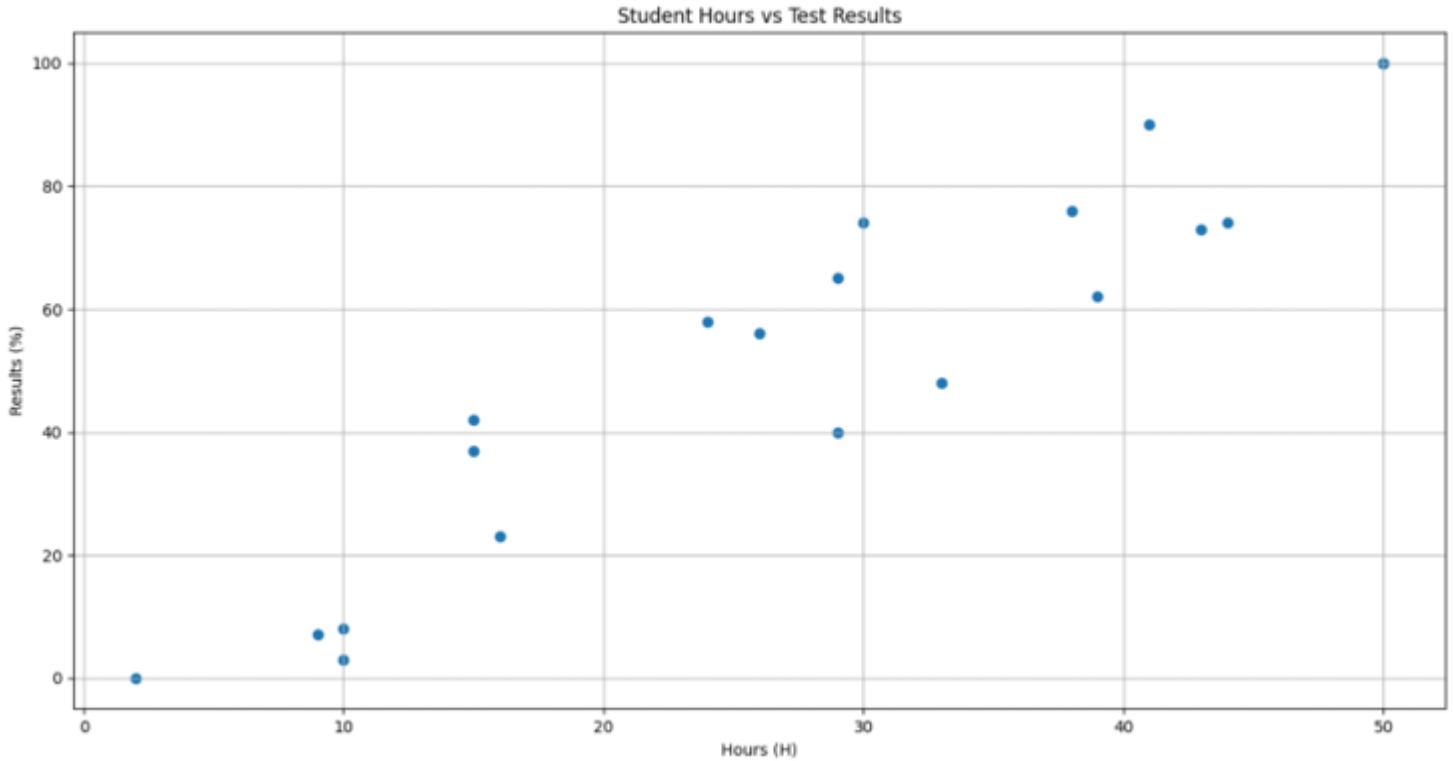
Linear Regression

Linear regression is an important tool to learn in data science for prediction and is a great foundation for more machine learning concepts. Let us first do a quick recap on the concept. Linear regression is based on linear functions for example the equation of straight-line `y = mx + c` is a simple yet powerful equation. Let us take a simple data set of the number of hours a student studies for and the results they achieve and plot it:

```
import matplotlib.pyplot as plt

hours = [29, 9, 10, 38, 16, 26, 50, 10, 30, 33, 43, 2, 39, 15, 44, 29, 41, 15, 24, 50]
results = [65, 7, 8, 76, 23, 56, 100, 3, 74, 48, 73, 0, 62, 37, 74, 40, 90, 42, 58, 100]

fig = plt.figure()
ax = plt.subplot(111)
ax.set_title("Student Hours vs Test Results")
ax.set_xlabel('Hours (H)')
ax.set_ylabel('Results (%)')
ax.grid()
plt.scatter(hours, results)
plt.show()
```



We can clearly see a linear trend showing if the student puts in more hours, they will get better marks however there is some variance. So, there must be a correlation between the two data sets. Now how can we derive a relationship from this dataset? That is where linear regression comes in.

So, to describe this relationship we can use the straight-line equation. Remember that `m` variable is called `slope` and the `c` variable is called `intercept`.

In the machine learning community, the `m` variable (the `slope`) is also often called the `regression coefficient`. The `x` variable in the equation is the *`input variable`* — and `y` is the *`output variable`*.

But in machine learning these x-y value pairs have many alternative names… which can cause some headaches. So here are a few common synonyms that you should know:

- `input variable` (x) – `output variable` (y)
- `independent variable` (x) – `dependent variable` (y)
- `predictor variable` (x) – `predicted variable` (y)
- `feature` (x) – `target` (y)

In any case let us now work out the straight-line equation in Python. For that we will be using our super useful Python library called `numpy` and its function called `polyfit`. All we need to do is add the following to our code:

```
model = np.polyfit(x, y, 1)
```

This executes the `polyfit` method from the `numpy` library that we have imported before. It needs three parameters: the previously defined input and output variables (x, y) — and an integer, too: 1. The latter number defines the degree of the polynomial you want to fit. Using polyfit, you can fit second, third, etc… degree polynomials to your dataset, too. (That's not called *linear* regression anymore — but *polynomial* regression.)

What's the math behind the line fitting. It used the ordinary least squares method (which is often referred to with its short form: OLS). It calculates all the errors between all data points and the model. Squares each of these error values. Then sum all these squared values. Then finds the line where this sum of the squared errors is the smallest possible value.

Now if you print what the model stores you should see:

```
2.015 x - 3.906
```

So 2.015 is the slope or regression coefficient and -3.906 is the intercept.

If a student tells you how many hours they studied, you can predict the estimated results of their exam. You can do this manually using the equation or using `numpy's` method `poly1d()` as shown below:

```
predict = np.poly1d(model)
hours_studied = 20
print(predict(hours_studied))
```

Output

```
36.38
```

Note: This is the exact same result that you'd have gotten if you put the `hours_studied` value in the place of the `x` in the `y = 2.01467487 * x - 3.9057602` equation.

R-Squared Calculation

There are a few methods to calculate `the accuracy of your model` namely the `R-squared ($R^2$) value`. I won't go into the math here, it's enough if you know that the R-squared value is a number between 0 and 1. And the closer it is to 1 the more accurate your linear regression model is. The equation for it is as follows:

$$R^2 = [\ (n\Sigma xy - (\Sigma x)(\Sigma y)) / (\sqrt{n\Sigma x^2 - (\Sigma x)^2} * \sqrt{n\Sigma y^2 - (\Sigma y)^2})\ ]^2$$

Unfortunately, `R-squared calculation is not implemented in numpy`... so we will be doing it by hand using the following code:

```
import numpy as np
hours = [29, 9, 10, 38, 16, 26, 50, 10, 30, 33, 43, 2, 39, 15, 44, 29, 41, 15, 24, 50]
results = [65, 7, 8, 76, 23, 56, 100, 3, 74, 48, 73, 0, 62, 37, 74, 40, 90, 42, 58, 100]
x = np.asarray(hours)
y = np.asarray(results)
sum_x = np.sum(x)
sum_x_2 = np.sum(np.square(x))
sum_y = np.sum(y)
sum_y_2 = np.sum(np.square(y))
sum_xy = np.sum(x*y)
print(f"sum_x = {sum_x}")
print(f"sum_x_2 = {sum_x_2}")
print(f"sum_y = {sum_y}")
print(f"sum_y_2 = {sum_y_2}")
```

```
print(f"sum_xy = {sum_xy}")
n = len(x)
print(f"n = {n}")
top = n*sum_xy - sum_x*sum_y
print(f"top = {top}")
bot_a = np.sqrt(n*sum_x_2 - np.square(sum_x))
bot_b = np.sqrt(n*sum_y_2 - np.square(sum_y))
bot = bot_a*bot_b
print(f"bot = {bot}")
R_2 = np.square(top/bot)
print(f"R_2 = {R_2}")
```

Output:

```
sum_x = 553
sum_x_2 = 19345
sum_y = 1036
sum_y_2 = 72414
sum_xy = 36814
n = 20
top = 163372
bot = 174378.40331875964
R_2 = 0.8777480188408425
```

Practically, we should just use a Python library called `sklearn` that can do this for us but doing the manual approach allows us to learn more about Python.

```
import numpy as np
from sklearn.metrics import r2_score

hours = [29, 9, 10, 38, 16, 26, 50, 10, 30, 33, 43, 2, 39, 15, 44, 29, 41, 15, 24, 50]
results = [65, 7, 8, 76, 23, 56, 100, 3, 74, 48, 73, 0, 62, 37, 74, 40, 90, 42, 58, 100]
```

```
model = np.polyfit(hours, results, 1)
predict = np.poly1d(model)

print(r2_score(results, predict(hours)))
```

Output

```
0.8777480188408424
```

To the plot the regression line over the data we will use `Matplotlib`. The extra line of code we need to do the regression line is to create a `numpy` array of values to input into the model. Since our `hours` range is from 0 to 50 we will use this as our data set upper and lower bonds using the method `linspace()`. The full code with all the functionality is shown below:

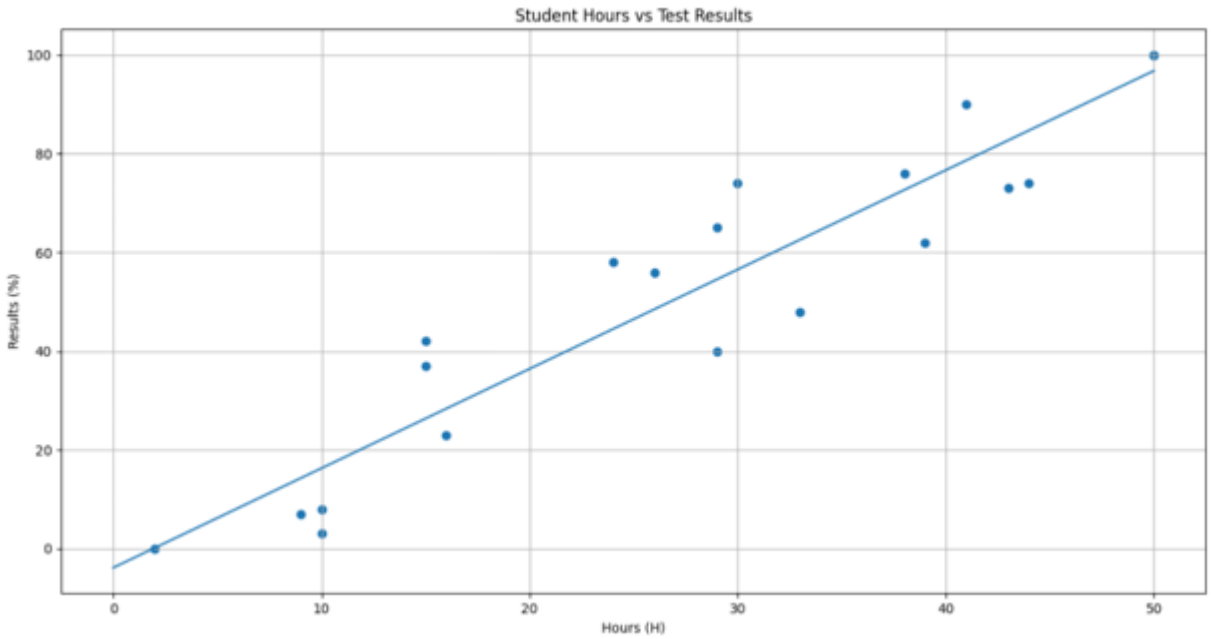
```
import numpy
import matplotlib.pyplot as plt

hours = [29, 9, 10, 38, 16, 26, 50, 10, 30, 33, 43, 2, 39, 15, 44, 29, 41, 15, 24, 50]
results = [65, 7, 8, 76, 23, 56, 100, 3, 74, 48, 73, 0, 62, 37, 74, 40, 90, 42, 58, 100]

mymodel = numpy.poly1d(numpy.polyfit(hours, results, 1))

regline = numpy.linspace(0, 50)

plt.title("Student Hours vs Test Results ")
plt.xlabel("Hours (H)")
plt.ylabel("Results (%)")
plt.grid()
plt.scatter(hours, results)
plt.plot(regline, mymodel(regline))
plt.show()
```



There are many other things which `numpy` can do, and not enough time in this course to cover all of them.

If you need to figure out how to do a particular task, have a look at the documentation and tutorial on the official `numpy` website:

https://numpy.org/doc/stable/ (https://numpy.org/doc/stable/)

Also, for still more specialised mathematical functions used in scientific applications you should have a look at the `scipy` library, which will be discussed later in the course.

https://www.scipy.org/ (https://www.scipy.org/)

`Scipy` is built on top of `numpy`, and designed to work well with it.

HPC

HPC stands for High-Performance Computing, and it refers to the use of supercomputers, clusters, and other specialized systems to perform advanced computation and data analysis at high speeds. These systems can handle large and complex datasets, perform complex simulations and modeling, and support parallel processing and other advanced techniques.

HPC can be a significant benefit to data scientists and academic researchers in science fields because it allows them to perform complex data analysis and simulations that would be infeasible or impractical on traditional computers. This can include things like modeling complex systems or analyzing large datasets, which can be crucial for making new discoveries and advancing scientific knowledge. Additionally, HPC can be used for machine learning and deep learning, which can help to improve the accuracy and efficiency of these technologies.

There are many ways that HPC can benefit academic researchers in science fields. Some examples include:

1. **Climate modeling:** HPC systems can be used to run large-scale climate simulations that can help researchers to understand and predict changes in the Earth's climate.
2. **Drug discovery:** HPC can be used to perform complex simulations of molecular interactions that can be used to identify new drug candidates and understand how they work.
3. **Astrophysics:** HPC can be used to simulate the formation and evolution of galaxies and other astronomical objects, which can help researchers to understand the origins of the universe.
4. **Genome analysis:** HPC can be used to analyze large genomic datasets and identify genes and genetic markers associated with diseases.
5. **Quantum chemistry:** HPC can be used to perform complex quantum mechanical simulations of chemical systems, which can help researchers to understand the properties of molecules and materials.
6. **Machine learning:** HPC can be used to perform large-scale machine learning and deep learning computations, which can help researchers to develop new algorithms and models for image recognition, natural language processing, and other applications.
7. **Neuroscience:** HPC can be used to run simulations of the brain and nervous system, which can help researchers to understand how the brain works and identify new treatments for neurodegenerative diseases.

These are just a few examples of the many ways that HPC can benefit academic researchers in science fields. The ability to perform complex simulations and data analysis at high speeds can enable researchers to make new discoveries and advance scientific knowledge in a wide range of fields.