Introduction to Scientific Python Workshop

T.M.Kinnear & Leon Schoonderwoerd

Contents

This Workshop is designed to outline an approach to the use and implementation of Python in your scientific research. It is a powerful tool for interacting with numerical simulation and data processing. It is portable and easy to adapt to specific usage and requirements. These are some of the features which have made it a successful language, highly used and widespread.

Python would be described as a high level language. We can provide code telling the computer what to do in fairly general terms, and the bits of the language and the hardware of the computer are already configured to manipulate that code into the actions necessary to provide the intended result.

It is also an 'interpreted' language. For some programming languages, all of the code must be written in advance, and then another program called a 'compiler' takes that code, and converts it into the actual instructions for the computer in one block. Interpreted languages don't do this in one go; every time a line of code is encountered, it compiles it there and then, and gives just that to the computer, then finds whether there is more code to look at. This is generally a lot slower than a code processed by a compiler, as there is time 'wasted' sending things back and forth, and doing things multiple times which need only have been done once if it knew how all of the code fits together. On the other hand, it means a dynamic interface can be provided, where you can type commands as you need them and have the computer process them as you decide what to do, seeing what results are along the way.

This workshop will assume a basic knowledge of the concepts of programming languages (variables/loops/conditionals/etc.)

Python can be used both interactively, with a 'command line' for typing lines of code as you go, and in a batch mode, where a file containing many lines of code is given to the interpreter. We will be looking at both uses, starting with the interactive mode.

Note

For future reference, any lines of code presented beginning with three 'greater than' symbols, `>>>`, indicate typing commands into the Python prompt in interactive mode. You do *not* need to type the 'greater than' symbols.

Python 2.X Difference

Note that two primary versions of Python are currently in use. One command in particular would be slightly different in Python 3.X versions. `print` is exclusively a 'function' in Python 3.X; meaning it **must** be called as follows:

```
print('Hello world')
```

as opposed to

```
print 'Hello world'
```

which is only valid in Python 2.X

1 Modules

1.1 NumPy

The module named NumPy provides access to a wide variety of mathematical routines and input/output processes to make it easier to write complex codes faster. It is recommended to have a look through the general capabilities which it offers, these and other details can be found on its website, http://www.numpy.org/. The website also has thorough definitions of all of the objects and methods which it provides, these can help in working out how to accomplish certain tasks, or better understand the wide variety of options some of the functionality provides.

Basic loading of a module, and some of the interaction with the NumPy module in particular are found in the following example:

```
>>> import numpy #load module
>>> print(numpy.__version__) #get version
'1.11.1'
>>> print(numpy.random.rand()) #make random number
0.062399692509412241
>>> print(numpy.sqrt(16.0)) #takes square root of 16
4.0
```

We can also store structures called 'arrays' using NumPy; these are very similar to lists, but are better to use for maths. This is due to the 'vectorisation' of maths applied to lists, which let us apply a process across all of the elements in the array in one command. See the following example.

```
>>> import numpy
>>> a = [1, 3, 5, 7, 20] #list
>>> b = numpy.array(a) #array
>>> print(a, type(a), a[1], type(a[1]))
[1, 3, 5, 7, 20] <type 'list'> 3 <type 'int'>
>>> print(b, type(b), b[1], type(b[1]))
[ 1  3  5  7 20] <type 'numpy.ndarray'> 3 <type 'numpy.int32'>
>>> #print(a**2) #would not work!
>>> print(b**2) #vectorised operation
[  1   9  25  49 400]
```

Another advantage of the NumPy module is its support for data processing. A file which is available to you through this workshop is `example.csv`, which contains several lines, each with a pair of comma separated values. We will now look at an alternative method for reading in those data, making use of the NumPy module.

```
import numpy
data = numpy.genfromtxt('example.csv',delimiter=',')
print(numpy.shape(data))
print(data)
```

The result should be,

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

In this case, the result is a 2D array (think of it as rows and columns) with all of the values from the file. The `numpy.shape()` function shows the dimensions of the 2D array, in other words, the first axis is 6 long, the second is 2 long.

We may not want a 2D array, we may prefer two 1D arrays to work with, each representing a column. NumPy permits us to do this, too.

```
import numpy
xs, ys = numpy.genfromtxt('example.csv',delimiter=',',unpack=True,dtype=int)
print(xs)
print(ys)
```

The result should be,

```
[ 1  1 -5  2  3  7]
[4 7 6 2 7 2]
```

The `unpack=True` argument tells it to break the 2D array into columns, as long as sufficient variable names are listed before the function (the `xs,ys`, for two columns). In addition, we have set an argument `dtype=int`, which forces the data type of the values to be integers rather than the default of floating point values.

NumPy's abilities do not end there! It also contains various statistical functions which are of great use. The following example illustrates some of the most commonly used.

```
import numpy
xs, ys = numpy.genfromtxt('example.csv',delimiter=',',dtype=int,unpack=True)
print(numpy.mean(ys), numpy.min(ys), numpy.max(ys), numpy.std(ys))
```

The result should be,

```
4.66666666667 2 7 2.13437474581
```

These produce the mean, minimum, maximum and the standard deviation of the '`ys`' column, respectively.

1.2 MatPlotLib

Another module available to us is MatPlotLib. MatPlotLib is a Python plotting library with functionality based on MatLab. The website documenting the project and all of its functions and commands can be found at http://matplotlib.org/. It will be used extensively later in this document, where more detail of its various features are covered.

1.3 SciPy

The module named SciPy provides access to a wide variety of primarily scientific routines. It is strongly connected to NumPy. It is recommended to have a look through the general capabilities which it offers, these and other details can be found on the website for the overall SciPy 'stack' and the SciPy module in particular https://www.scipy.org. In addition, it has thorough definitions of all of the objects and methods which it provides, these can help in working out how to accomplish certain tasks, or better understand the wide variety of options some of the functionality provides. One of its common uses is the variety of fitting routines provided via its '`.optimize`' routines.

2 Tools

2.1 Plotting

This section on plotting is built around the use of the MatPlotLib module, in particular a subset of routines, `matplotlib.pyplot`, though there are other ways of performing the same tasks with alternative modules. We can start by looking at a simple example.

```
import matplotlib.pyplot as plt
xs = [1,2,3,4]
ys = [4,3,2,1]
plt.plot(xs,ys,'ro')
plt.show()
```

This will result in the plot displayed in Figure 1. It is extremely simplistic, but we have taken a simple series of data points split across two columns (`xs` and `ys`) and produced a basic plot. Note that the axes are unlabelled and the axis ranges are inappropriate (the first and final datapoints are

on the plot, but not clear as they are at the very edge. Note that the string `'ro'` on the plotting line is a formatting string describing how to present the points. The `'r'` indicates it should be red (try others such as `'g''b''y''k''m''c'`) and the `'o'` indicates it should be a circular marker (try others such as `'.''x''+''s''d'`).

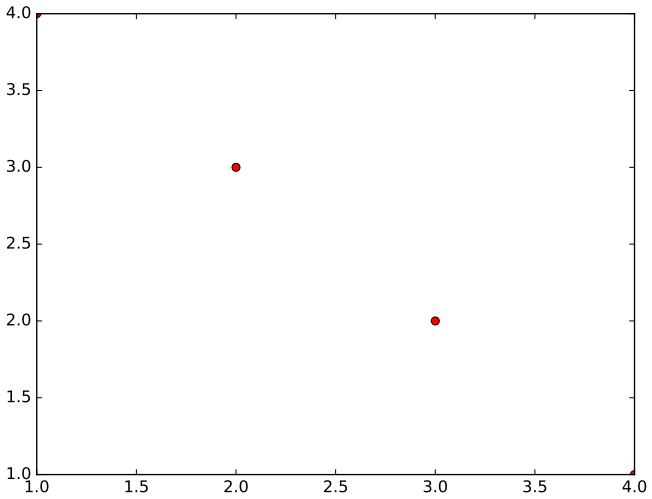

Figure 1: Figure produced from the execution of the first example code in Section 2.1

Before examining how to correct the shortcomings of this particular plot, let's examine some more plot types.

One typical example of a plot which can be conveniently constructed in scripts like Python, but which are awkward in software like Excel, is that of plotting histograms. The following example shows a sample 1D data set, and creates a histogram of those data.

```
1 import matplotlib.pyplot as plt
2 vals = [-2, -5, 1, 1, 0, -1, -2, -2, 0, 0, -3, 2, 1, -2, -2, -1, 2, -1, 0, 1, 1, 0, 1, 1,
  ↪  1, 0, 3, 0, -1, 1, -1, 5, 2, -3, 3, 0, 0, 0, 1, -1]
3 plt.hist(vals)
4 plt.show()
```

The histogram produced from this code can be seen in Figure 2.

We can also generate data ourselves, rather than specify it. As it happens, the integers presented in the example producing Figure 2, appearing manually in the example code, represent samples from a normal distribution. NumPy can be used to illustrate the behaviour of a normal distribution using its random number sampling routines.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 vals = np.random.normal(loc=0.0,scale=2.0,size=1000)
4 plt.hist(vals)
5 plt.show()
```

The plot from this code is displayed in Figure 3. This example uses `np.random.normal()` to sample from a normal distribution. The optional arguments represent properties of that distribution, and what is returned from the function itself. `loc=0.0` specifies that the 'location' (the mean of the distribution) should reside at zero; `scale=2.0` indicates the the standard deviation should be `2.0` and `size=1000` instructs the routine to return 1,000 values from the distribution. As with before, the command `plt.hist()` will plot the 1D dataset returned on a histogram, and `plt.show()` instructs Python to display it to screen.

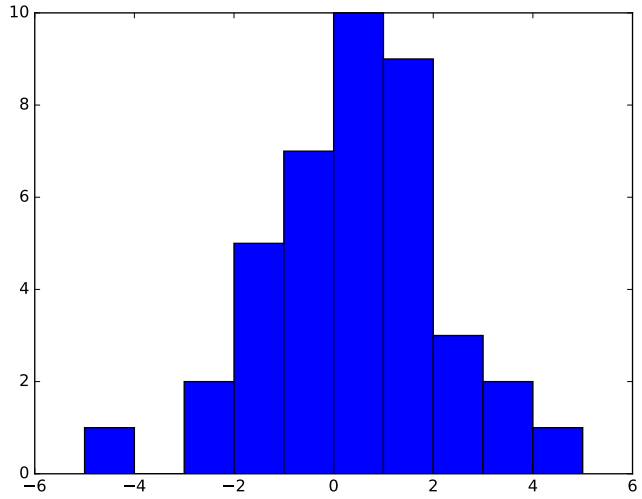

Figure 2: Figure produced from the execution of the second example code in Section 2.1, illustrating the production of histograms.

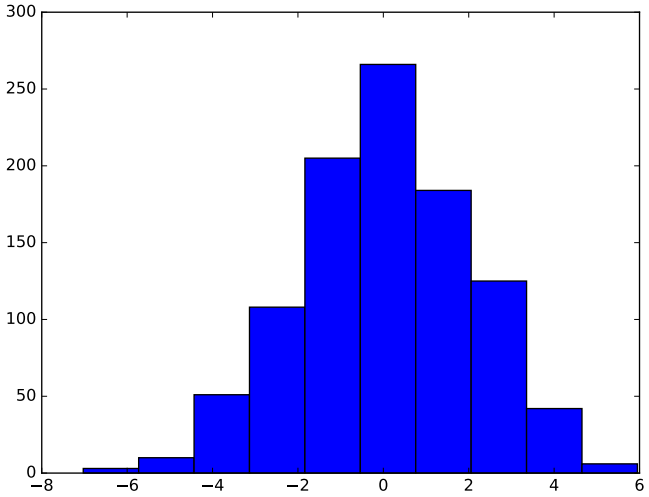

Figure 3: Figure produced from the execution of the third example code in Section 2.1, illustrating the generation of random samples from a distribution and presentation as a histogram.

Once more assuming the same `example.csv` file as in the NumPy section, we can begin making use of both NumPy and MatPlotLib to extract and plot data from a file. We will plot the data from that file as a simple line graph initially.

```
import numpy as np
import matplotlib.pyplot as plt
xs, ys = np.genfromtxt('example.csv',delimiter=',',dtype=int,unpack=True)
ys2 = ys**2
plt.plot(xs,ys,'ro')
plt.plot(xs,ys2,'go')
plt.xlabel('my x axis')
plt.ylabel('my y axis')
plt.savefig('myplot.pdf')
plt.show()
plt.clf()
```

This will produce a plot with two sets of data points. One is the raw data from the file, the other is each of the y values squared.

The result is the plot shown in figure 4.

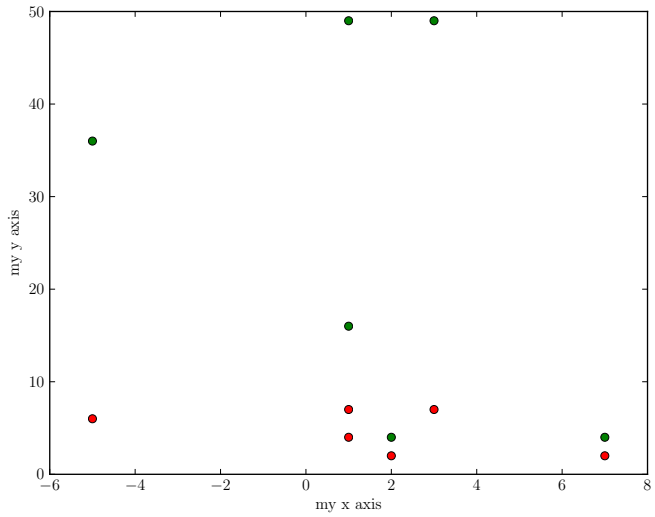

Figure 4: Figure produced from the execution of the fourth example code in Section 2.1

We can also make use of the functionality of NumPy and MatPlotLib, in particular the use of vectorised operations, to generate values from arbitrary functions to plot.

```
import numpy as np
import matplotlib.pyplot as plt
ts = np.arange(0,2.0*np.pi,0.01)# or...
ts = np.linspace(0,2.0*np.pi,1000)
xs = 16 * np.sin(ts)**3
ys = 13 * np.cos(ts) - 5 * np.cos(2 * ts) - 2 * np.cos(3 * ts) - np.cos(4 * ts)
plt.plot(xs,ys,'r-',lw=5)
plt.savefig('shape.pdf')
plt.clf()
```

Any idea what the functions presented in this code will produce? The output is displayed in Figure 5. Firstly, we have used `numpy.arange(start,stop,step)` or `numpy.linspace(start,stop,quantity)` to generate a sequence of numbers between 0 and 2π for the array `ts`. Then we have a parametric function for `xs` and `ys` with a combination of mathematical operations to produce our coordinate pairs, applying the equations to the entirety of the `ts` array.

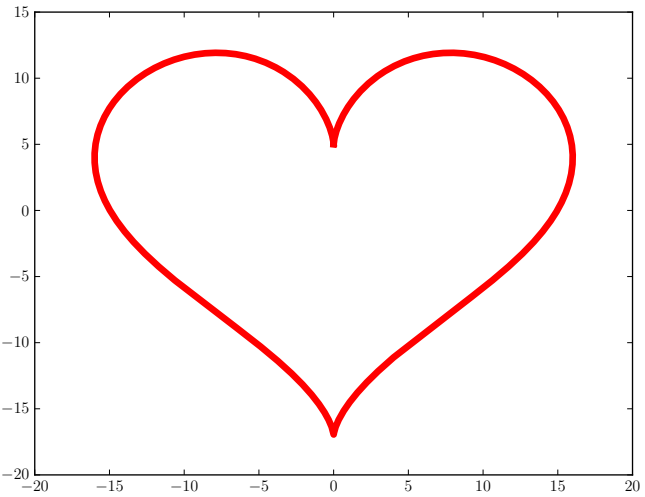

Figure 5: Figure produced from the execution of the example code featuring the `def` statement in Section 2.1

We can also write functions using the `def` statement. The following example performs the same task as the previous one, but instead of all being in one 'monolithic' code, the section which produces the x, y datapoints for the heart shape itself has been separated out into a function, which can be called whenever needed.

```
import numpy as np
import matplotlib.pyplot as plt

def heart(resolution=100):
    ts = np.linspace(0,2.0*np.pi,resolution)
    xs = 16 * np.sin(ts)**3
    ys = 13 * np.cos(ts) - 5 * np.cos(2 * ts) - 2 * np.cos(3 * ts) - np.cos(4 * ts)
    return xs,ys

xs,ys = heart(1000)

plt.plot(xs,ys,'r-',lw=5)
plt.savefig('shape.pdf')
plt.clf()
```

2.2 Exercise: plotting

Exercise 1

Rather than giving you some contrived example to practice on, we suggest you select an example from your own research to make a plot. Try imagining what you would want the plot to look like, then comb through the documentation at http://matplotlib.org/ (feel free to use Google and other sources as well!) to figure out how to achieve your vision. This is how *real* programmers do it, too!

2.3 File Handling

2.3.1 Basic File Handling

The default Python function for opening a link to a file is `open()`. You can see in the previous sections pertaining to NumPy that its own `.loadtxt()` or `.genfromtxt()` functions can be useful for reading in data from files, especially where the file format is very regular. However, there will be circumstances where you may not wish to interact with a file this way, or it would otherwise be inconvenient. One way of opening a file and reading the content from it in native Python might be the following.

```
f = open('myfile.txt','r')
for line in f.readlines():
    print('I am doing a thing with the line', line)
f.close()
```

This is not ideal and should be discouraged. There are issues with file handling, the specifics of which can be found in more detail elsewhere, which must be dealt with carefully. The most common issue is failure to properly close or release a file object, such that the file becomes corrupted or remains locked for subsequent operations. The preferred method, to deal with the major issues encountered in working with open files, is as follows.

```
with open('myfile.txt','r') as f:
    for line in f.readlines():
        print('I am doing a thing with the line', line)
```

This has removed the need to have `f.close()` and accommodates a large variety of 'exception handling', where the `with` statement effectively understands the context of many of the error messages or issues which might occur. Correctly closing a file, and in particular closing a file correct if something goes wrong, can be very important. In the first example, if anything happened during the `print` lines, the code would never execute the `f.close()` line, and the file object would remain open. This might close automatically if the Python kernel ends, but particularly when working with IDEs with persistently running kernels (such as Spyder) the link to the file has not been closed, and issues could be caused. This could be especially troublesome with large files and/or writing to files; even more so if this is part of a much larger/longer task where losing output could result in the need to run hours or days of calculations again!

This process can also be used to open and handle multiple files in one go, which can be particularly convenient.

```
with open('myinputfile.txt','r') as fin, open('myoutputfile.txt','w') as fout:
    #doing stuff with both files available to be read from and written to.
```

2.3.2 Awkward Data Formats

Sometimes we will be presented with, or otherwise be in possession of, data files which do not have a convenient format. There may be mixtures of datatypes, or issues with linebreaks, or any number of other oddities. Non-straightforward data files can be awkward to read in using the quick NumPy commands, and we may still wish to avoid a long-winded process of writing a manual read-in script.

Fortunately, some fairly awkward files can still be processed with a few additional options. Considering a file with the following content:

```
a,3,1,3.5563
b,2,1,3.6123
a,3,0,7.1234
b,4,1,7.2344
c,6,1,2.1355
```

It contains a text column, an integer column, a column with zeros and ones representing boolean values and a floating point column. A useful way to deal with data like this (or, for other reasons, any data) is using a custom datatype structure. This is illustrated in the following example.

```
import numpy as np
import matplotlib.pyplot as plt
mydtype = np.dtype([("StringField", np.str_, 512), ("IntegerField", int), ("BooleanField",
 ↪  bool), ("FloatField", np.float64)])
data = np.genfromtxt('strangedata.csv',delimiter=',',dtype=mydtype)
print(data)
```

The data type is set up on the line specifying "`mydtype =`". `numpy.dtype()` permits one to define a new data type using a list of 'tuples' containing a name for the component and datatype for the component and, if necessary, a size. The first is presented as `("StringField", np.str_, 512)`, its name is `"StringField"` (as the first column can be stored as strings), the datatype used is the specific `numpy.str_` type, though this could just be the default Python `str` type. Finally, its defined as having a length of 512 characters. This is more than necessary for this example, but is done to illustrate the option. The other fields are defined as their respective data types (not requiring lengths for simple numeric data).

The data from the file are then read in using a single `numpy.genfromtxt()` command, using the optional argument and value `dtype=mydtype` to specify that the type we have just defined describes the data to be read in.

The `print` command will produce an array of each line of the file split into its relevant components. More usefully, we can add the following lines to the previous code.

```
print(data['StringField'])
print(data['IntegerField'])
```

the result of these two lines would be,

```
['a' 'b' 'a' 'b' 'c']
[3 2 3 4 6]
```

where we have selected a 'column' of data based on its name. We can further reference individual rows combined with the column names. So `data[2]['StringField']` would be the third row (remembering Python zero-base numbering system), of the string column.

If we further add the following lines, we can perform even more tasks:

```
selected = data['FloatField'] > 3.6
print(data[selected]['StringField'])
selected = np.logical_and(data['FloatField'] > 3.6,data['StringField'] == 'a')
print(data[selected]['IntegerField'])
```

In this case the `selected` array is constructed of boolean values representing whether the statement `data['FloatField'] > 3.6` is true or not. NumPy permits indexing of arrays using boolean arrays, i.e. selecting elements out of one array where the boolean array is `True` and ignoring those where it is `False`. This means that `data[selected]['StringField']` will be all values in the string column, where its corresponding value of the float field was greater than 3.6. The next command makes a different selection, based on two properties at once, using NumPy's `logical_and` function.

For certain data, it can sometimes be necessary to convert the raw data into a different form before storage. For the current example, let's imagine that the `a`'s, `b`'s and `c`'s are meant to be 1, 2 and 3 respectively. We can do the following, using a converter function to accomplish this.

```
import numpy as np
import matplotlib.pyplot as plt

```

```
4  def my_converter_function(value):
5      if (value == 'a'):
6          return 1
7      elif (value == 'b'):
8          return 2
9      else:
10         return 3
11
12 mydtype = np.dtype([("FirstField", int), ("IntegerField", int), ("BooleanField", bool),
   ↪  ("FloatField", np.float64)])
13 myconverters = {0:my_converter_function}
14 data =
   ↪  np.genfromtxt('strangedata.csv',delimiter=',',dtype=mydtype,converters=myconverters)
```

What this does is that the `myconverters = {0:my_converter_function}` statement defines a dictionary (another PYTHON construct which can be quite useful, but which hasn't been covered in full here); when given to the read in function as the optional argument and value `converters=myconverters`, it instructs the zeroth column (first column of the file) to be passed through the '`my_converter_function`' function before storage. The function itself is set up to give back the number which should correspond to each letter. As it happens, there are better ways of writing such a function, but this is a clear manner.

These tools combined should permit approaching the majority of types and formats of files, to be able to extract data in a usable fashion.

2.4 Fitting

ONE extremely useful facility for any data analysis is the ability to fit a function to measured data. This occurs in practically every field, as it is a ubiquitous quality that data should be associated with some manner of theory.

We will be covering how one can fit (almost) arbitrary functions to our data, and use already designed fitting algorithms to accomplish this. One further module is introduced in this section, SCIPY. SCIPY accompanies NUMPY for scientific data processing and analysis. In particular, we will use several functions within the `.optimize` set of SCIPY routines.

First, we will generate some pretend data. The following sequence of commands is accompanied by a series of plots illustrating the result of each step.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  xs = np.arange(-50,50,0.1)
5  ys = 0.2 * xs**3 - 3.0 * xs**2 - 230.0 * xs + 20
6  plt.plot(xs,ys,'k.')
7  plt.clf()
```

this is displayed as Figure 6, showing a cubic polynomial in the range -50 to 50 with some arbitrarily selected coefficients.

These data are now modified using the following commands,

```
8  ys += np.random.normal(scale=np.max(np.abs(ys))*0.1,size=len(ys))
9  plt.plot(xs,ys,'k.')
10 plt.clf()
```

The initial line (8), adds a random perturbation to every data point using the NUMPY `random.normal()` function. As the `size` argument for the normal distribution is set to be the same size as the `ys` array, they are the same length, and NUMPY will automatically add them together pairwise (first with first, second with second, etc.). The result of the next plot is shown in Figure 7 where all of the data points have been jostled vertically.

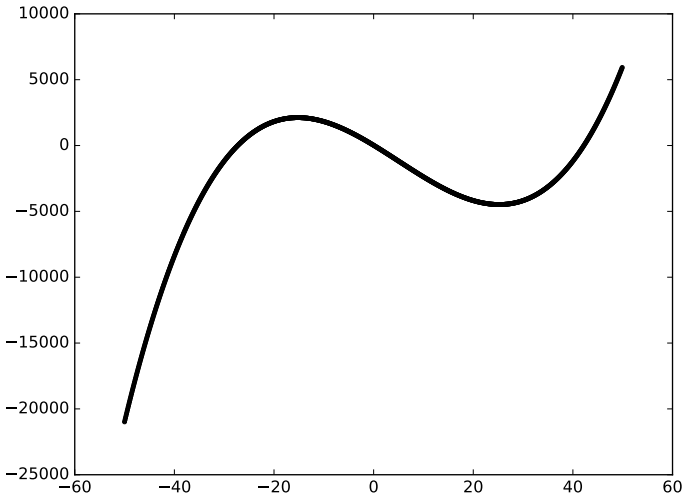

Figure 6: Plot of initial 'perfect' data for simulating experimental data in Section 2.4.

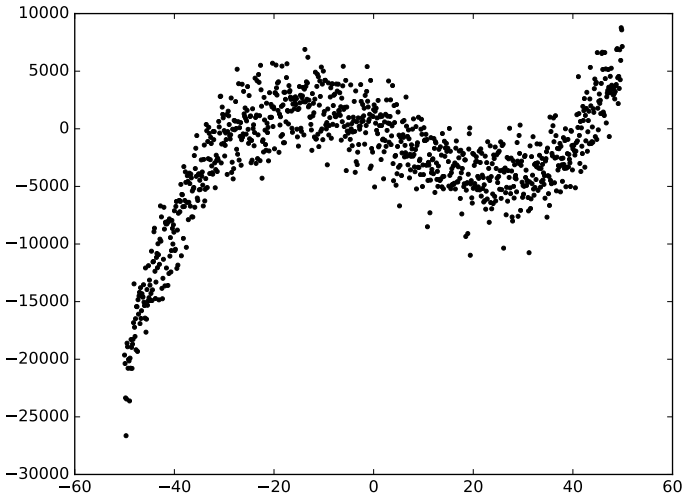

Figure 7: Plot of randomly perturbed data from Section 2.4.

Now, we will attempt to fit a new polynomial to the jostled data from scratch, with no knowledge of the original coefficients being provided. For this, we need to import the SciPy module.

```
11 import scipy as sp
12 myfit = sp.polyfit(xs,ys,3)
13 print("({0[0]:0.2f}) x^3 + ({0[1]:0.2f}) x^2 + ({0[2]:0.2f}) x +
    ↪  ({0[3]:0.2f})".format(myfit))
14 fit_function = sp.poly1d(myfit)
15 plt.plot(xs,ys,'k.')
16 plt.plot(xs,fit_function(xs),'r-')
17 plt.clf()
```

Having imported SciPy, we make use of its `.polyfit()` function. `.polyfit()` can be provided with the x and y values of the data, and the order of polynomial to fit (3 in the example shown), and it will return an array of the coefficients for the fitted polynomial. These values are displayed by the `print` command using a formatting statement; producing (for these particular data):

```
(0.20) x^3 + (-2.98) x^2 + (-228.16) x + (12.77)
```

These fitted coefficients being fairly close (given the random variation added to our data) to our originals of 0.2, -3.0, 230.0 and 20. SciPy also provides a function to easy produce a polynomial data function out of coefficients (`scipy.poly1d(coeffs)` where `coeffs` needs to be an array of the coefficients), this is stored under a new name of `fit_function` on line 14. Providing the x coordinates to this function will cause the generation of the corresponding y coordinate for each x. So lines 15 and 16 plot the original data, and the fitting line respectively. This produces the plot displayed in Figure 8.

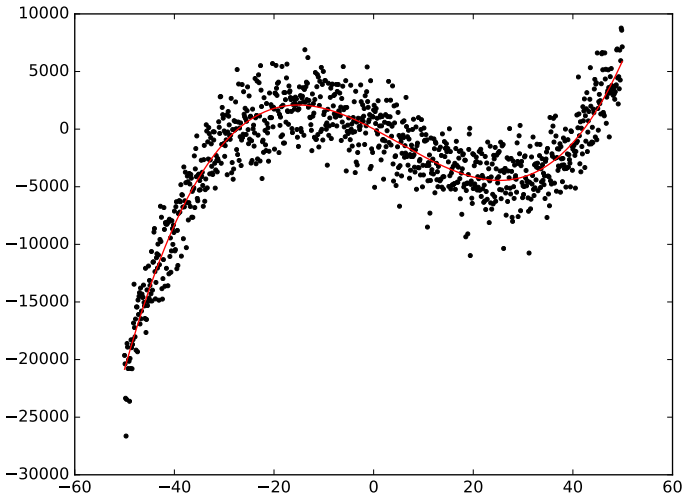

Figure 8: Plot of randomly perturbed data plus the polynomial fit from `scipy.polyfit()` from Section 2.4.

This means that we are easily able to fit polynomials. However, so can we in Excel. One feature which is especially useful is the ability to define our own function to fit. The following is an example of a sparsely populated dataset from an arbitrary function.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  xs = np.linspace(-4*np.pi,4*np.pi,100)
5  ys = 2.0 * np.e**(-0.1*np.abs(xs)) * np.sin(xs)
6  ys += np.random.normal(scale=0.1,size=len(ys))
7  plt.plot(xs,ys,'k.')
8  plt.show()
9  plt.clf()
```

Figure 9 shows the result of this code, which is a damped oscillation shape, but as described, fairly sparsely defined (there is notable space between each point).

To fit an arbitrary function, we are usually required to define a custom function which the fitting routine is aiming towards, with some particular format of input arguments.

```
9  from scipy import optimize
10
11 def myfunction(xs, a, b):
12     return a * np.e**(-b*np.abs(xs)) * np.sin(xs)
13
14 popt, pcov = optimize.curve_fit(myfunction,xs,ys)
15
```

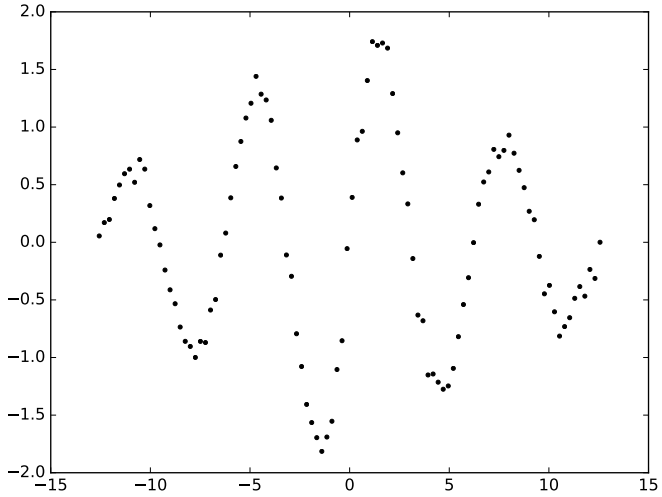

Figure 9: Plot of the arbitrary function from Section 2.4.

```
16 print(popt)
17 print(pcov)
18
19 plt.plot(xs,ys,'k.')
20 many_xs = np.linspace(-4*np.pi,4*np.pi,1000)
21 plt.plot(many_xs,myfunction(many_xs,*popt),'r-')
22 plt.show()
23 plt.clf()
```

First, we import the subset of SciPy, the `.optimize` routines. This provides the `.curve_fit()` routine, which uses a least-squares method to fit a functional form to a set of data. Lines 11 and 12 define the function itself, which has arguments in the pattern starting with the data for the x values, then then each of the parameters defining the behaviour of the function. When passed to `.curve_fit()` it is these parameters which will be fitted. Recalling the original function plotted, these were fixed as 2.0 and 0.1 for our 'experimental' data.

The call to `.curve_fit()` itself occurs on line 14. It returns two arrays by default. One which is the optimised parameters, and one which is the covariance matrix. The exact meaning of the values in the covariance matrix is complicated, but it can be thought of as describing uncertainties in the fit. The values returned by the `print` commands for this example were:

```
[ 2.07946146  0.10403299]
[[ 1.21160175e-03  9.31470879e-05]
 [ 9.31470879e-05  1.14368464e-05]]
```

This seems a reasonable fit, given that our starting values for `a` (first value in the array) and `b` (second value in the array) in our original function were 2.0 and 0.1 respectively.

Finally, we plot the original data on line 19. Then we generate a finer array of x locations (denser than the original sparse dataset) on line 20. This is used in our `myfunction` function along with the values of the fitted/optimised parameters to generate a curve representing our fitted function (the asterisk before the name of the array in the argument list requests that the array be unpacked first and given as two separate arguments, rather than giving the array itself as a single argument). The result of the overall plot can be seen in Figure 10. A good extension to this would be adding a figure legend. This can be done by adding the argument '`label='my series label'`' to the `plt.plot()` commands, and adding `plt.legend()` before `plt.show()`.

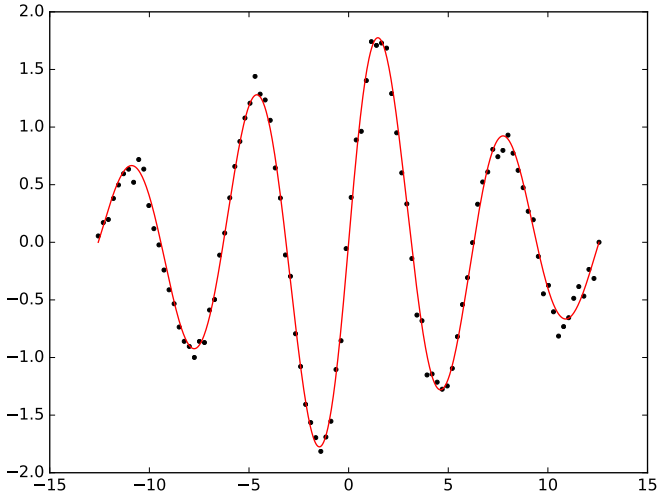

Figure 10: Plot of the arbitrary function along with fitted curve for the function from Section 2.4.

2.5 Exercise: file handling and fitting

Exercise 2

Among the files you have been given via a USB drive or GitHub, there is a file called `weather_data.txt`, consisting of five columns. These are an index, a timestamp and two data columns, with temperature and cumulative rainfall (i.e. the amount of rain fallen since the last timestamp). The data is collected over 24 hours.
Try reading in the data, then making appropriate fits and/or plots to visualise the data. You are of course free to do this as you see fit, but some hints are given below.

1. We expect the temperature data to roughly follow a single sinusoidal oscillation, mediated by noise. A scatter plot with fit like in Fig. 8 should yield good-looking results here.

2. The rainfall data is probably best visualised using a histogram. Experiment with the histogram parameters to e.g. find the right bin size. Can you come up with a probability distribution for the bins, and fit this to your plot?

3 Applications

3.1 Orbital Mechanics (Dynamic Simulation)

THERE are several approaches to modelling orbital mechanics/orbital dynamics. The method briefly described here is an intuitive one, but not usually the most effective for the timescales typical to the scenarios being examined (where 'symplectic integrators' are preferred). However, this gives sufficient approximations for a 'toy model' to examine the idea of dynamical simulations. This is one in a category of what are named 'N-body codes'.

This model will calculate the instantaneous behaviour of the system, and then say that, for a sufficiently small step in time, this behaviour does not change drastically. This means that the instantaneous behaviour can describe the motion for that entire step in time. Once that motion is evaluated, we find the instantaneous behaviour for that new configuration, and so on.

The initial setup requires a set of particles, where any given particle i has a starting position, $\mathbf{r}_i = (r_{i,x}, r_{i,y})$, velocity $\mathbf{v}_i = (v_{i,x}, v_{i,y})$ and mass m_i. Note that this example is working in two dimensions, but can easily be modified to all three.

The force on any particle i due to another particle j through their gravitational attraction is given by,

$$\mathbf{F}_{ji} = G\frac{m_i\, m_j}{|\mathbf{r}_{ji}|^2}\hat{r}_{ij}, \tag{1}$$

where the subscript notation is, for example $\mathbf{F}_{ji}$ the force of particle j acting on particle i and $\hat{r}_{ij}$ is the unit vector pointing from particle i toward particle j.

This means that if we have a collection of particles, we can look at each of them in turn, sum the forces acting on them due to the other particles, and find a net force for the given particle $\mathbf{F}_i$. Once this net force calculation is performed on all particles, we can evaluate the expected motion of the particles. Obtaining an acceleration is easy, as,

$$\mathbf{F} = m\mathbf{a}, \tag{2}$$

for constant mass. This means that,

$$\mathbf{a}_i = \frac{\mathbf{F}_i}{m_i} \tag{3}$$

To change the acceleration into motion is where we need to assume a step in time which is small enough that the acceleration can be said to be 'constant'. Obviously, even the slightest movement of the particles changes the acceleration, as it is a function of their position. We are positing a sufficiently small change in time that this is negligible compared to our desired accuracy. We will call this time Δt.

With this in mind, the new velocity of the particle will be,

$$\mathbf{v}_i^{\text{new}} = \mathbf{v}_i^{\text{old}} + \mathbf{a}_i\Delta t, \tag{4}$$

and once this new velocity is calculated, the new position on the same assumptions will be,

$$\mathbf{r}_i^{\text{new}} = \mathbf{r}_i^{\text{old}} + \mathbf{v}_i^{\text{new}}\Delta t. \tag{5}$$

One thing to note is that, in this quick example, there have been some 'handwave-y' assumptions in the sequence of the updates to both velocity and position. A better method for this is known as the 'leapfrog' method, if you are interested, it may be worth looking in to.

All of this combined leads to a set of three critical equations defining the behaviour of the particles:

$$\begin{aligned}
\mathbf{a}_i &= \frac{G}{m_i}\sum_j\left(\frac{m_i\, m_j}{|\mathbf{r}_{ji}|^2}\hat{r}_{ij}\right)\\
\mathbf{v}_i^{\text{new}} &= \mathbf{v}_i^{\text{old}} + \mathbf{a}_i\Delta t\\
\mathbf{r}_i^{\text{new}} &= \mathbf{r}_i^{\text{old}} + \mathbf{v}_i^{\text{new}}\Delta t.
\end{aligned} \tag{6}$$

Exercise 3

Attempt to implement the described orbital mechanics simulation! The main things to consider is how information is to be stored, the treatment of the vector quantities (it might be easiest to decompose into x and y). The other thing to consider is starting quantities and what constitutes a 'sufficiently small' time step. As a hint, a recommended maximum might be no larger than $1/100^{\text{th}}$ of the smallest orbital period.

Exercise 4

An extremely similar simulation can be set up to simulate Rutherford Scattering, between an incident alpha particle and a stationary gold nucleus. In this instance, the acceleration equation takes the form:

$$\mathbf{a}_i = \frac{1}{4\,\pi\,\epsilon_0\,m_i}\sum_j\left(\frac{q_i\, q_j}{|\mathbf{r}_{ji}|^2}\hat{r}_{ij}\right), \tag{7}$$

where q is the charge on each particle.
This scenario is most easily approached using 'atomic units', the important definitions for which are that charge is measured in units of magnitude of electron charge (such that $q_{\mathrm{Au}} = +79$ and $q_\alpha = +2$), mass is measured in units of electron mass (such that $m_\alpha = 7294.3$), and distances are in units of Bohr Radii (where characteristic interaction distances might be 0.001). Velocities can be calculated based on alpha particle energies for a given source. Americium has an alpha particle energy of approximately 5.5 MeV.

3.2 Monte Carlo (Stochastic Simulation)

Monte Carlo methods broadly rely upon use of randomisation to find approximations of solutions to problems. These can be especially useful for scenarios which are simple to set up, but complex in their interactions and difficult to optimise. In addition, they general scale especially well for parallelisation. Most algorithms on parallel systems moved to n cores/threads do not run n times faster, due to overheads in the threads needing to intermittently update one another on what is going on. The previous example of simulation, N-Body codes, can scale particularly poorly.

Monte Carlo methods cover a broad range of actual algorithms. The common factor is the use of repeated random sampling.

Exercise 5: approximating π

Write a script that approximates π by the following algorithm:

1. Generate N pairs of random numbers $(a, b) \in [0, 1] \times [0, 1]$.
2. For each pair, calculate the distance of this point to the origin $(0, 0)$.
3. Approximate π as $\pi \approx 4\frac{N(r<1)}{N}$, where $N(r < 1)$ is the number of points within a distance 1 from the origin.

To generate random numbers, write

```
import random
```

at the start of your script, and use

```
x = random.random()
```

to assign a random number in the interval $[0, 1)$ to `x`. You can read the documentation at docs.python.org/3/library/random.html for more information about Python's built in random number functionality.
Try to think about how this problem can be approached using loops, which may be more intuitive; and how it can be approached using the vectorised operations facilitated by NumPy array structures, which will be substantially faster.
It is worth considering that this method is basically one option for numerical integration, which is the process which is effectively being performed.

3.3 Root Finding by Bisection

One category of problems covered by numerical methods are those of 'root-finding'. This can be simplified as finding a way to determine where a function crosses a particular value; usually phrased such that the value in question is zero. This sits close to another category, being 'minimisation' problems, where the aim is to find the parameter, or combination of parameters, which make a function as small as possible.

The example to cover here is based on finding the value of a square root numerically. This may seem like a trivial operation. However, it is difficult to approach using only basic mathematical operations capable of giving direct numerical answers. Start with the following equation:

$$x = \sqrt{\alpha}, \tag{8}$$

we have α, and we wish to determine x. However, what actual operations do we perform to do so?

There is seldom ever a simple, perfect solution for deciding which algorithm should be used to solve any general problem. Usually, any particular algorithm will have advantages and disadvantages, and each problem will have features which play differently with efficacy of the algorithms.

Rephrasing our equation as

$$x^2 - \alpha = 0 \tag{9}$$

reforms the problem into the form of root finding. For a given α there will be some value of x (in this case, two values, a positive and negative version) for which Equation 9 is True.

Approach here will be the method named '*bisection*', being 'to cut in two'. It is a type of 'bracketing' method, and there are certain guidelines for when this is a valid method which can be used. In general, it is a fairly effective, general purpose algorithm; being simple to approach and usually very easy to implement for a given problem.

The idea behind the method is that we start off with knowledge of a particular range of values of x certain to contain a single solution. This is (fairly) easy for the specified problem; the square root of α. As we know that the two solutions (positive and negative) are symmetric around zero, we only need to find one. This needs to be between bounds we will name x_{low} and x_{high}.

We will be evaluating an intermediate equation which is not quite Equation 9:

$$x'^2 - \alpha = k. \tag{10}$$

Where k is the residual, which we want to reduce to (or as close to) zero, to have recreated Equation 9.

This leads to the following key steps for our algorithm:

1. Set x_{high} to α or 1, whichever is larger.
2. Set x_{mid} to $(x_{\text{low}} + x_{\text{high}})/2$ ($= x_{\text{high}}/2$ in our example, as x_{low} is implicitly zero).
3. Set Δx to $(x_{\text{low}} + x_{\text{high}})/4$ ($= x_{\text{high}}/4$ in our example, as x_{low} is implicitly zero).
4. Evaluate Equation 10 to get k.
5. If k is negative:
   - Set x_{mid} to $x_{\text{mid}} + \Delta x$.
6. If k is positive:
   - Set x_{mid} to $x_{\text{mid}} - \Delta x$.
7. Set Δx to $\Delta x/2$.
8. Repeat from step 4 until a value of x_{mid} is obtained which is sufficiently accurate.

The algorithm as a whole is remarkably simple for how comparatively effective it is. It is ultimately just trial an error with decreasingly small steps.

For a simple coded implementation on PYTHON, with $\alpha = 9$, running ten iterations was too short to be timed using `time.time()`. Running 1,000,000 iterations takes approximately 0.36 s. Effectively about 360 ns per iteration. However, not nearly this many iterations would be needed. To the double floating point variable precision ($\approx$ 16 decimal places), we reach the solution of $x = 3$ exactly at 53 iterations, or 19 µs.

Exercise 6: Square root by bisection

Code your own example of the bisection algorithm for the square root problem.

Exercise 7: Square root by Newton-Raphson

Finding the square root based on bisection is not particularly efficient. A preferred algorithm is the Newton-Raphson method. In this method, the next estimate x_{i+1} in the sequence is given by the location that a line given by the tangent to the function at the current estimate x_i crosses the x-axis. This line is calculated using the value of the function and its gradient. It can be summarised as:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \tag{11}$$

or in the case of the square root example, using equation 9, specifically,

$$x_{i+1} = x_i - \frac{{x_i}^2 - \alpha}{2\,x_i}, \tag{12}$$

where α is the value of which we are attempting to determine the square root.
Attempt to implement this algorithm, and determine the number of steps required to reach the same precision as the bisection method.

3.4 Numerical Integration and Differentiation

THERE are a variety of circumstances in which one needs to determine the integral or derivative of numerical data. This may be experimental data, or data from a function which is either awkward or impossible to approach analytically. The concept of differentiation and integration as taught in schools, using discrete examples of Δx and Δy and the trapezium rule form the basis for simple implementations of these methods.

The main approach for differentiation is using finite differences. These can be thought of in three categories, forward-, backward- and central-difference approaches. Forward difference is given by $f(x+\Delta x)-f(x)$; backward by $f(x)-f(x-\Delta x)$ and central by $f(x+\Delta x/2)-f(x-\Delta x/2)$. These might seem to only have marginal differences (hah!) but have importance when considering certain circumstances. One circumstance which may become most obvious will be boundary conditions. Specifically, if using any of these finite difference methods, what happens for the very first and very final points. Some methods will calculate the derivatives for the midpoints in x, giving derivatives which are misaligned with the original data, but which do not make assumptions about the behaviour outside the bounds of those data. Some will use various assumptions such that the locations in x of the derivative values match the original set of x points.

Assuming the mid-point values are fine, we can say that for a set of points x_i and y_i for i between 0 and $N-1$, the derivatives at the locations $i+\frac{1}{2}$ can each be given by

$$\left.\frac{\mathrm{d}y}{\mathrm{d}x}\right|_{i+\frac{1}{2}} = \frac{y_{i+1}-y_i}{x_{i+1}-x_i} \tag{13}$$

and will sit at locations $x_{i+\frac{1}{2}} = (x_{i+1}+x_i)/2$.

Basic numerical integration can be performed as per the trapezium rule, but can suffer similar issues in exact implementation as numerical differentiation. In particular, whether the integration is considered to be the definite integral between x_i and x_{i+1} or whether it is the indefinite integral at a specific x.

Using a similar approach as with the differentiation example but, we can calculate the integral between x_0 and x_k with,

$$\int_{x_0}^{x_k} y(x)\mathrm{d}x = \sum_{i=0}^{k-1} \left(\frac{y_{i+1}+y_i}{2}\right)(x_{i+1}-x_i). \tag{14}$$

Note that for uniformly spaced data, all $x_{i+1} - x_i$ are the same, and can be replaced with Δx.

Exercise 8

Among the files you have been given via a USB drive or GitHub, there is a file called `velocities.csv`, consisting of two columns. These are time and velocity. Try reading in the data and then using numerical integration and differentiation to determine the acceleration of the object with time and distance travelled.

4 Finale

You should now have access to a variety of tools which permit useful tasks to be performed with Python. Python has immense capacity for processing data, analysing data, generating simulations, coding interactive GUIs and much more. Far more, in fact, than can be included in such a workshop as this.

A wide variety of further resources are available, both physically and on the Internet. I would encourage you to consult as many of these as possible and gain knowledge through exploring applications of coding and how to approach data analysis computationally.

4.1 Further Reading

This workshop only gives an opportunity for a very coarse overview of what can be done with Python or, for that matter, computational methods in general. Depending on your research interests, you may have cause to use some of the specifics featured here, or otherwise build upon them. However, your particular interests may lie primarily outside of the applications covered here. Some suggestions of further reading, or useful resources for several topics follow.

- Data processing:
  Some coverage of data processing is featured here, but there are a wide variety of approaches to the topic. One popular module for reading, analysing and processing data is 'Pandas'. Documentation for which can be found at `pandas.pydata.org`. It specialises in unifying several aspects covered in this workshop, in particular merging features of NumPy and MatPlotLib with its own composite commands. It also deals with 'data frames' which behave somewhat like a NumPy custom datatype array structure, with convenient connections to statistical analysis and plotting methods.

- Machine Learning:
  Machine learning is a growing topic. There are a variety of categories of machine learning, possible the most high profile of which is that of 'neural networks'. Many of these used in research make use of 'TensorFlow', documentation of which can be found at `tensorflow.org`. Another is PyTorch, which can be found at `pytorch.org`.

- Interfacing with existing programs/services:
  Due to its popularity, modules exist for interacting with a huge range of software and systems. Scripts for interacting with web-based systems, in particular, have a great deal of utility. Many web systems use systems such as Django (`www.djangoproject.com`) or host SQL databases which can be communicated with using Python mysql, sqlite3 or similar.