ENGG1003 - Lab 2

Brenton Schulz

1 Library Basics

This lab will be your first official introduction to using Python *libraries*.

A library is a collection of *functions* (blocks of code referenced by name with specific input, functionality, and output) which perform tasks beyond those "built in" to the Python language.

For example, Python has no built in way of calculating square roots so this mathematical operation is implemented in the `sqrt()` function from the `math` or `numpy` libraries.

Libraries can also contain data in addition to functions. For example, the `math` library (used in the task below) contains the numerical constant π, accessed by using the text `pi`.

Task 1: Math Library Example

Read through Section 1.3 of the textbook (`https://link.springer.com/chapter/10.1007/978-3-030-16877-3_1#Sec9`).

Be sure to observe the (intentional!) error which results from running the first code example

The full documentation for all `math` library functions can be read here: `https://docs.python.org/3/library/math.html`

2 Importing Library Functions

Python allows for several different ways to *import* libraries, each with their own *syntax* and behaviour.

Task 2: Importing Libraries

Read through Section 1.4 of the textbook `https://link.springer.com/chapter/10.1007/978-3-030-16877-3_1#Sec10`

Note the difference between:

- `from library import things`
- `from library import *`
- `import library`
- `import library as name`

Section 1.4.5 lists the libraries used by the textbook. Some of them (like `math`) are installed by default while others need to be explicitly installed.

3 Installing Libraries

Normally you would only install the libraries needed for a particular project. It is also best practice to limit the installation of a library to a *virtual environment*. The reasons can be complex, but you should be aware that libraries are often under constant development and their behaviour can change as updates are released. Containing each Python project inside a virtual environment means that each project can use different versions of each library. This is done so that code written in the past will continue to work as new updates are released because that project's virtual environment only contains libraries known to work with that code.

Task 3: Installing Libraries

Install the `numpy` and `matplotlib` libraries in your PyCharm project.

To do this, click on the "Terminal" tab (to the left of the "Python Console" tab) and execute the following commands:

- `pip install numpy`
- `pip install matplotlib`

Test that the installation was successful by running `import numpy` and `import matplotlib` in the Python console.

4 An Introduction to Plotting and Vectorisation

This section requires `numpy` and `matplotlib` so if you skipped the previous Task please go back and finish it.

Plotting requires a new programming concept: the *array* (also known as a *vector* or, in Python a *list*). So far a single variable has been a single value (eg: `g = 9.8`). An array is an ordered collection of numbers all referred to by the same variable name.

In the task below you will see a line of code which reads:

```
t = np.linspace(0, 1, 1001)
```

This line of code uses the `linspace` library function to create an array of 1001 numbers linearly spaced between `0`, and `1`.[1].

Task 4: Your First Array

Run the following code in the Python Console and observe the resulting array of values:

```
import numpy as np
x = np.linspace(0,10,11)
print(x)
```

Note how the numbers `0` to `10` are printed. All of these values are stored as a "group" inside the variable `x`. The trailing decimal points indicate that the values are *floating point* numbers - more on these later, but a `float` *datatype* is required to store numbers with both integer and fractional components.

In Python (and most programming languages) you can only plot *numbers*. This is a subtle but crucial point: Python (specifically the `matplotlib` library) doesn't understand how to draw an "equation" or "formula". It is only able to join lines between an ordered set of points on a 2D plane.

[1] Complete `linspace()` documentation is here: `https://numpy.org/doc/stable/reference/generated/numpy.linspace.html`

To plot the graph of an equation you must first write other Python code which generates a set of points on that graph then pass the resulting arrays to `matplotlib`.

To do this "simply" (in few lines of code) a new idea is required: *vectorisation*. Vectorisation is a large topic, but for now know that using an array in a mathematical expression repeats the calculation for *every* value in the array.

Note the output of the previous task: the variable `x` contained 11 values. The next example will demonstrate *vectorising* some arithmetic using that result.

Task 5: Vectorisation Example

Building on the code in the previous task, copy the code below into a script and run it. Observe that the result of the $y = 2 * x$ expression is also an array. This is a *vectorised* expression because it operates on an entire array in one line of code. All 11 values in `x` have been multiplied by 2 and the results stored in the variable `y`.

```
import numpy as np
x = np.linspace(0,10,11)
print(x)
y = 2*x
print(y)
```

With those fundamental principles explored, now is a good time to complete reading and executing textbook Sections 1.5 and 1.6.

Task 6: Section 1.5

Read through and execute all code in textbook Sections 1.5 to learn how to create basic plots using the *vectorisation* techniques shown above.

Link: `https://link.springer.com/chapter/10.1007/978-3-030-16877-3_1#Sec16`

Section 1.6 of the textbook covers a few examples which create "complete" plots with titles, labels, legends, etc. Since the full code can look overwhelming at first the following tasks show minimalistic examples of drawing multiple curves on one plot and creating *subplots*.

Task 7: Multiple curves in One Plot

As seen in Section 1.5, when drawing a plot two functions must be called:

- `plt.plot()` - to draw data to an "internal buffer"
- `plt.show()` - to actually draw the plot to the screen

In order to draw multiple lines on the same plot your code can call `plt.plot()` more than once. Every set of data sent to `plt.plot()` will create a new curve that gets drawn to the plot window when `plt.show()` is called.

Run the following code as an example. It plots two straight lines.

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

x = np.linspace(-2,2,100)
y1 = 2*x+5
y2 = -0.5*x+2

plt.plot(x,y1)
plt.plot(x,y2)
plt.show()
```

Lastly, you can add a grid to the plot by running:

```
plt.grid()
```

before calling `plt.show()`.

Task 8: Subplots

The creation of *subplots* allows multiple separate graphs to be drawn in the same plot window. This is done by calling `plt.subplot()` *before* calling `plt.plot()`. The `plt.subplot()` function selects which plotting area inside the complete plot window the next call to `plt.plot()` will draw to.

The `plt.subplot()` function has a syntax of `plt.subplot(r,c,n)` where `r` and `c` specify the shape of a grid of plots in terms of the number of rows, `r`, and columns, `c`. The parameter `n` specifies which subplot should be drawn to (indexed from 1) with following calls to `plt.plot()`.

Read, copy, then run the following code:

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

x = np.linspace(-2,2,100)
y1 = 2*x+5
y2 = -0.5*x+2

plt.subplot(2,1,1)
plt.plot(x,y1)

plt.subplot(2,1,2)
plt.plot(x,y2)

plt.show()
```

The first call to `plt.subplot()` creates a 2x1 grid of plots and selects the first plot. The call to `plt.plot(x,y1)` below it then draws this line to plot number 1.

The second call to `plt.subplot()` continues to specify the 2x1 grid but selects plot number 2. The call to `plt.plot(x,y2)` therefore draws to the second subplot area.

Finally, `plt.show()` draws both graphs to the screen.

Task 9: Section 1.6

Read and execute the more detailed plotting examples in Section 1.6.
Link: `https://link.springer.com/chapter/10.1007/978-3-030-16877-3_1#Sec17`

5 Further Exercises

Task 10: Plotting Polynomials

With what you have learned so far, write a Python script which initialises three variables: `a`, `b`, and `c`, and plots the resulting parabola from the equation:

$$y = ax^2 + bx + c \tag{1}$$

Experiment with the range of `x` values used so that the graph produced for `a = 2`, `b = 5`, and `c = -2` clearly shows both roots (x-axis crossings). Estimate their values from the plot and compare your numbers to the exact values of:

$$x = -\frac{5}{4} - \frac{\sqrt{41}}{4} = -2.8508$$
$$x = -\frac{5}{4} + \frac{\sqrt{41}}{4} = 0.35078$$

Task 11: Simultaneous Equations

Write a Python script which allows a user to solve the following simultaneous equations:

$$2x + 4y = 3 \tag{2}$$
$$x - 2y = 1 \tag{3}$$

Hint: Re-arrange both equations to make y the subject and plot them both on the same graph. The "solution" is the point at which the lines intersect.

Task 12: Plotting Trigonometric Functions

The `numpy` library includes a vast array of mathematical functions from trigonometry to logarithms. Write a Python script which plots $\sin(x)$, $\cos(x)$, and $\tan(x)$ on the same graph from 0 to 4π.

Note that:

- These functions assume *radian* input
- Since they are from the `numpy` library you need to reference them appropriatly. Review Section 1.4 for details, or use `import numpy as np` followed by `np.sin()` (etc).

You will encounter a problem: $tan(x)$ approaches $\pm\infty$ and, by default, totally obscures sin() and cos()!

To fix this, read about the `ylim()` library function (from `pyplot`) and restrict the y-axis to the range $[-1.5, 1.5]$. The documentation is here: `https://stackabuse.com/how-to-set-axis-range-xlim-ylim-in-matplotlib/`

Task 13: Linear Interpolation

There are many instances in science and engineering when data needs to be *interpolated*. Informally, this is the process of predicting a quantity's value somewhere between two (or more) data points.

In this task you will be using the *two-point formula* to interpolate between two data points. You can imagine these data points to be measurements of temperature / brightness / sound intensity / pressure / etc as a function of time.

The two-point formula states that the equation of a line passing through two points (x_1, y_1) and (x_2, y_2) is:

$$y = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1. \tag{4}$$

The value of y at some given x can then be estimated. For the purposes of interpolation x will always be in between the given data points. ie, assuming $x_2 > x_1$:

$$x_2 > x > x_1. \tag{5}$$

Note that y is only an *estimate* of the true value. You will often see this notated as $\hat{y}$, especially in statistics.

And just to really hammer the point home: x is any point; (x_1, y_1) and (x_2, y_2) are fixed points, and y is the value to be calculated given (x_1, y_1), (x_2, y_2), and x.

Given the above, write a Python script which implements the following pseudocode:

```
BEGIN
  Create variables for x1, y1, x2, y2, and x
  Estimate the value of y at x using the linear interpolation formula
  PRINT y
  Give the user confidence in the answer by plotting a graph
    showing the three points.
END
```

See Fig. 1.2 of the text (and surrounding context) for how to plot individual points.

Extension: Given any value of (x_1, y_1) and (x_2, y_2) plot the line which passes through these two points between $x = -10$ and $x = 10$. You will need to create your own x array with `linspace()`. Plot both the resulting line and the two input points with an $\star$ marker.