Week 11: Plotting and NumPy arrays

Intended learning outcomes

By the end of this class, you will be able to:

- use the Matplotlib module to plot graphs;
- create and edit Numpy arrays.

1 Basic plotting with Matplotlib

Matplotlib (short for mathematics plot library) is a Python module that allows you to do some very advanced plotting. Here we will look at some of its basic features. Run the following program. It should display a graph.

```
1 import matplotlib.pyplot as plt
2
3 x_values = [0, 1, 2, 3, 4]
4 y_values = [0, 4, 3, 4, 3.5]
5
6 plt.plot(x_values, y_values)
7
8 plt.show()
```

Line 1 imports the relevant part of the Matplotlib module. The `as plt` part means that to call a function from the module, we only have to type `plt.function()` instead of `matplotlib.pyplot.function()`. This is the standard convention for Matplotlib.

Lines 3 and 4 define the x- and y-coordinates of the points we are going to plot. Note they are stored in two separate lists. The points that are plotted are $(0, 0)$, $(1, 4)$, $(2, 3)$, $(3, 4)$ and $(4, 3.5)$.

Line 6 defines (but does not display) a plot of the points we have defined. By default, the points will be joined with blue lines.

Line 8 displays the plot we have defined. As we will see, you might want to do various things to your plot before displaying it.

Now change line 6 to each of the following in turn, running the program at each go.

```
plt.plot(x_values, y_values, color='red')
plt.plot(x_values, y_values, color='red', marker='o')
plt.plot(x_values, y_values, color='red', marker='o',
         linestyle='None')
plt.plot(x_values, y_values, 'r')
plt.plot(x_values, y_values, 'ro-')
plt.plot(x_values, y_values, 'ro')
```

Hopefully you have noticed that the last three lines are just more cryptic versions of the first three lines. There are many options available for the plots. Don't worry about them now, but they can be found at `https://matplotlib.org/2.0.2/api/pyplot_api.html`.

Now we can add a second graph. Insert the following code into your program somewhere before the `plt.show()` command.

```
y_values_2 = [2, 0, -1, 4, 2.5]
plt.plot(x_values, y_values_2, color='black')
```

When you run the program you should see both graphs on your plot.

Next, include some labels by adding the following lines before the `plt.show()` command.

```
plt.title("A very basic plot")
plt.xlabel("x")
plt.ylabel("y")
```

2 Basic function plotting with lists

You have seen above that we can get Python to plot a graph if you have a list of x-values and a corresponding list of y-values. To plot a function, such as $\sin(x)$, you will need to create these two lists and you can proceed as you did when tabulating values of $\sin(x)$ on a previous sheet.

We are going to plot the graph for x from $x_{\min}$ to $x_{\max}$ and we are going to split the interval from $x_{\min}$ to $x_{\max}$ into N equal width subintervals with endpoints $x_0, x_1, x_2, \ldots x_N$ so $x_0 = x_{\min}$ and $x_N = x_{\max}$. These are equally spaced, so the spacing is $(x_{\max} - x_{\min})/N$. This means that x_i is given, for $i = 0, \ldots, N$, by

$$x_i = x_{\min} + i\frac{(x_{\max} - x_{\min})}{N}.$$

We can use this to build up our lists of x-values and y-values,

$$[x_0, x_1, \ldots, x_N] \quad \text{and} \quad [f(x_0), f(x_1), \ldots, f(x_N)],$$

term by term, using a `for` loop and the `list.append()` method. We can then get Matplotlib to plot the points.

Here it is implemented to plot $\sin(x)$ for x from 0 to 2π, using 8 intervals (i.e. 9 points).

```
1  import math
2  import matplotlib.pyplot as plt
3
4  x_min = 0
5  x_max = 2*math.pi
6  NO_OF_INTERVALS = 8
7
8  x_values, y_values = [], []
9
10 for i in range(NO_OF_INTERVALS + 1):
11     x = x_min + i * (x_max - x_min) / NO_OF_INTERVALS
12     x_values.append(x)
13     y_values.append(math.sin(x))
14
15 print("\nx:", x_values, "\n\ny:", y_values, "\n")
16
17 plt.plot(x_values, y_values)
18
19 plt.show()
```

Run the program. You should see the two lists printed out and you should see a jaggedy version of the sin function plotted. The two lists are just printed out so you can see that the `for` loop has done its job. This is not necessary for the running of the program, but printing out things like this can be useful for debugging.

For instance, if you are unsure what the loop is doing you can indent line 15 by four spaces to make it part of the loop, and so print out the lists each time the loop goes round. You can also add a four-spaces indented `print(i)` to line 14 to see how far round the loop you are.

Line 15 is an ugly line and once you have seen what it does you can "comment it out" by putting a hash before it.:

```
# print("\nx:", x_values, "\n\ny:", y_values, "\n")
```

Now change line 6 so that the number of intervals is 20. Run the program. What does the graph look like? Now change it to 100 and run the program again. Any better? In general, you will use enough points so that the graph looks smooth.

Add the following to line 18. Notice the difference that it makes to the plot.

```
plt.gca().set_aspect("equal")
```

This makes the x- and y-axes be plotted with the same scale. This is often a very important thing to be able to do, especially if you want geometric figures like squares and circles to look correct.

Exercise 11.1.

- Modify the program to also plot $\cos(x)$, in red, on the same graph.
- Add a title and labels on the axes.

3 Function plotting with NumPy arrays

We have seen that we can use Python lists with Matplotlib to plot graphs. However, lists are not the best thing to use; the code can become a bit of a mess, as we saw above. Using **NumPy arrays** is much cleaner and quicker. The downside is that you have to learn about NumPy arrays! NumPy stands for Numerical Python and it is a very important module for numerical computations.

We will work here back in the console. It is traditional to abbreviate numpy to np in functions, so first import the NumPy module in the following way:

```
>>> import numpy as np
```

The main variable type used by NumPy is the NumPy array. At first glance, NumPy arrays look like Python lists, but they behave quite differently.

```
>>> array_1 = np.array([1.1, 2.2, 3.3])
>>> print(array_1)
>>> type(array_1)
>>> list_1 = [1.1, 2.2, 3.3]
>>> print(list_1)
```

Notice how a list is printed with commas in between and a numpy array has no commas. A NumPy array is more like a vector than a list, in that operations you perform on it don't tend to alter the length, unlike operations on lists. You know that `+` concatenates (joins together) two lists and `list * 5` repeats a list five times. These have different meanings for NumPy arrays, as operations are usually performed *on each individual element*. Compare the following:

```
>>> print(list_1 + list_1)
>>> print(list_1 * 5)
>>> print(array_1 + array_1)
>>> print(array_1 * 5)
```

What happened in the last two cases? We say that operations are **performed pointwise** on NumPy arrays. This has the advantage that we can apply mathematical functions to every element of a NumPy array at the same time, whereas with lists we had to use a **`for`** loop and go through each element. The important thing to remember is that we have to apply the NumPy version of the function: use `np.sin()` rather than `math.sin()`.

```
>>> print(np.exp(array_1), "\n", np.log(array_1))
```

In the first case we have taken the exponential of each entry, in the second case we have taken the logarithm of each entry.

Remember that to plot a function in Matplotlib we need a list, or array, of x-values, and the corresponding list, or array, of y-values. With NumPy it is easy to create an array of equally spaced points between two values: the command `np.linspace(x_min, x_max, N)` will create a NumPy array of `N` equally spaced points from `x_min` to `x_max`, which *includes both endpoints*.

```
>>> print(np.linspace(0, 1, 11))
>>> print(np.linspace(1, 3, 5))
```

All that we need now is to apply a function to the resulting array and feed it into Matplotlib. This means that we can make our plot of $\sin(x)$ with x from 0 to 2π using the much cleaner and compact program.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x_values = np.linspace(0, 2*np.pi, 100)
5 plt.plot(x_values, np.sin(x_values))
6 plt.show()
```

Exercise 11.2. Modify the program so that it also plots the function $\cos(x)$, in red, on the same graph.

If you wish to further explore Matplotlib, have a look at the nice introduction at `https://github.com/rougier/matplotlib-tutorial`.

Homework

The group project will be launched this week, so there is no assessed homework for the next lab.

1. Finish off the sheet.

2. (*Quick review.*) What is the output of the following? Type them in to check.

(i)
```
>>> import numpy as np
>>> print([1, 2] * 2)
>>> print(np.array([1, 2]) * 2)
>>> print([1, 2] ** 2)
>>> print(np.array([1, 2]) ** 2)
```

(ii)
```
import matplotlib.pyplot as plt
plt.plot([0,1], [5, 4])
plt.show()
```

3. (*Optional.*) Using the parametric equation of a circle, get Python to draw a circle. (We will look at this next week.)

4. (*Optional.*) The `plt.plot()` function requires a list of x-coordinates and a list of y-coordinates. Sometimes you will have the data as a list of pairs of coordinates and will want to convert to the form required by `plt.plot()`. Write a function `separate_coordinates()` that does this conversion: the function will take a list of pairs of numbers, such as `[[2, 3], [0, 0], [5, 1]]` and will return a list of the first coordinates and a list of the second coordinates, such as `[2, 0, 5]` and `[3, 0, 1]`.