CSC120H Lab 8

1 Objectives

- Practise using external Python modules
- Practise loading data from a file
- Practise plotting graphs with `matplotlib`

Note: This lab uses external modules that do not come installed with Python. The Lab computers have these modules installed, but your computer likely doesn't. If you haven't yet installed these, please work on this lab on the Lab computers first, and then at the end of the lab you will have a chance to install the modules on your own computer.

2 Making a basic line graph

You will start by making a basic line graph. Create a file called `lab8_part2.py`. At the top, write the statement:

```
import matplotlib.pyplot as plt
```

This imports the plotting module from matplotlib and gives it the name `plt` as an easier to use shorthand for the module to use in your script.

Now, let's plot a line using individual points. Just like you would plot a line on graph by hand, you need to indicate to Python the location of the points by giving their horizontal and vertical positions (such as on the x and y-axis).
For each axis, add code that creates a list that corresponds to that axis:

```
x = [0, 1, 2, 3, 4]
y = [2, 5, 7, 3, 9]
```

Each index of the `x` list corresponds to a value at the same index of the `y` list. These two lists represent the points `(0,2), (1,5), (2,7), (3,4),` and `(4,9)` on the plotted graph. It makes sense that these two lists must be of the same length, so that each x coordinate has a corresponding y coordinate.
Now we will plot the points on the graph using the plotting module.
First, we use the `plot` method to plot the points, giving as arguments the two lists:

```
plt.plot(x, y)
```

If you run the code now, you won't see the plot come up, because you must indicate to Python that you want to display it. Later we will see how we can modify the plot before showing it. For now, simply add the line:

```
 plt.show()
```

Run the file and you should see a plot pop up in a separate window. If a plot doesn't come up, ask your TA for assistance.
Looking at the plot, you'll notice a few things: Although the x-axis starts at 0, the y-axis starts at 2, since that's the lowest number in the y-coordinate list. Also, there are no axis labels, and no title for the graph. Luckily, these are all elements that you can set yourself!

Modifying plot elements

Let's write some code that modifies these elements of the plots.
First, let's add some labels to the axes. The methods that add axis labels to the plot are `plt.xlabel` and `plt.ylabel`. Since these values don't represent anything specific, we will use the strings `'x'` and `'y'` as the axis labels.
Put the following code **above** the `plt.show()` call. This lets you set up and modify the plot the way you want to before finally showing it on the screen. You can put the code below the `plt.plot(x, y)` call.

```
plt.xlabel('x')
plt.ylabel('y')
```

Run the program again and notice the the axis labels now appear (with the y-axis label flipped to read vertically).
Now, let's add a title (put this code under the label code):

```
plt.title('Line Graphs')
```

Run the program again to see the title.

Notice that we are using the `plt` module that we imported as a variable that can directly access the current plot. There is, at this point, only one graph attached to this plot figure. Let's see how we can put two line graphs on the same plot figure.

Directly under the `x` and `y` lists, create two new lists:

```
x2 = [0, 1, 2, 3, 4]
y2 = [6, 3, 9, 8, 5]
```

Now, under `plt.plot(x, y)`, add another call to the `plot` method:
`plt.plot(x2, y2)`

Run the program again. You should now see two different line graphs on the same plot figure.
Often it's good to have a legend to indicate what each of these lines means. First, you must name each of the line graphs by adding a third optional argument `label` to the `plt.plot` calls:

```
plt.plot(x, y, label='Line 1')
plt.plot(x2, y2, label='Line 2')
```

We can then add the legend simply by writing:
`plt.legend()`
before showing the plot. This will find the best place for the legend, where it is hopefully out of the way of the lines themselves. Run the program to see it.

One more modification: we can set the range of the y axis to start from 0 and go to 10. Put this code before showing the plot:

```
plt.ylim(0, 10)
```

Notice that the y-axis now starts at 0 and goes to 10.
Try setting the range to go from 5 to 10. Notice that the graph will get cut off if you do this.

Saving plot images Often you will want to save your plot images to your computer to open and use later without having to run your code again. We can save plot figures as images by using the method `plt.savefig` to save a png image file:

```
plt.savefig('line_graphs.png')
```

You can put this line of code after the show() call if you want, since you can both show a figure and save it to a file. If you look in the folder where your Python file exists, you should now see the file `line_graphs.png` appear as well. You can also save a PDF file by changing the file extension to be `line_graphs.pdf`.

3 The `numpy` module; Creating sequences

In this section, we're going to look at another Python module called `numpy`. This module provides numerical methods and data types that can help make calculations and plotting much easier.
Create a new file called `lab8_part3.py`. At the top of this file, write:

```
import numpy as np
```

In the last section, we created the list for the x-values by using a list with a sequence from 0 to 4, `[0, 1, 2, 3, 4]`. Obviously, if we have a hundred points on the x-axis, we don't want to have to manually write out a hundred values.
We will instead use a method from `numpy` called `arange` that can easily create a sequence of numbers for us. Write the following code under the import statement:

```
x = np.arange(0, 10)
print(x)
print(type(x))
```

Look at the result in the shell. The first print statement prints out the sequence, from 0 up to but not including 10. Although it looks like a `list`, this is actually a type that `numpy` creates called a 'numpy array'. For our purposes, you can basically treat it as a list. (Note: this looks like it is similar to the `range` that we use for looping over indexes, but numpy is optimized for large sequence lists and plotting, so we use it for this purpose instead of using `range`).

We can change the 'step size' of our sequence by adding an additional argument:

```
x2 = np.arange(0, 10, 2)
print(x2)
```

This will print a sequence starting from 0 up to but not including 10, stepping over every other digit.

Creating sequences practise

Use the appropriate calls to `np.arange` to make the following sequences.
Print out your solutions one after the other in the Python file so that they all appear in the shell.

1. Create an increasing integer sequence from 3 up to but not including 12.

1. Create an increasing integer sequence from 5 up to and including 34, with a step size of 3.

2. Create a decreasing integer sequence from 20 down to but not including 3 (note that you will need a step size of `-1`)

3. Create a sequence from 5 to 14, in steps of 0.2.

4. Create a sequence from 54 to 12, in steps of 6.

5. Create a sequence from 0 to 30, in steps of 2.
6. Create a sequence from 0 to 30, in steps of 1.
7. Create a sequence from 0 to 30, in steps of 0.1.
8. Create a sequence from 0 to 30, in steps of 0.01.

Plotting a function

Since we can create sequences of numbers for the x-axis of a plot, we can easily create plots of math functions. Let's plot a quadratic function.
First, let's add the import for matplotlib under the one for numpy:

```
import matplotlib.pyplot as plt
```

Now, make an x-axis array:

```
quad_x = np.arange(0, 10)
```

To make the y-axis array for the quadratic function, we can use `quad_x` and operate on it like a quadratic math function $y = x^2$ would be expressed:

```
quad_y = quad_x ** 2
```

Now, plot the points:

```
plt.plot(quad_x, quad_y)
plt.show()
```

Plotting a sine function

Try doing the same thing but instead of a quadratic function, create a sinusoidal function.
`numpy` has a built-in function that returns the sine of a number - you can access it using `np.sin()`.
Try it using this x-axis array:

```
sin_x = np.arange(0, 50)
```

Create the appropriate y-axis array and then plot the graph.

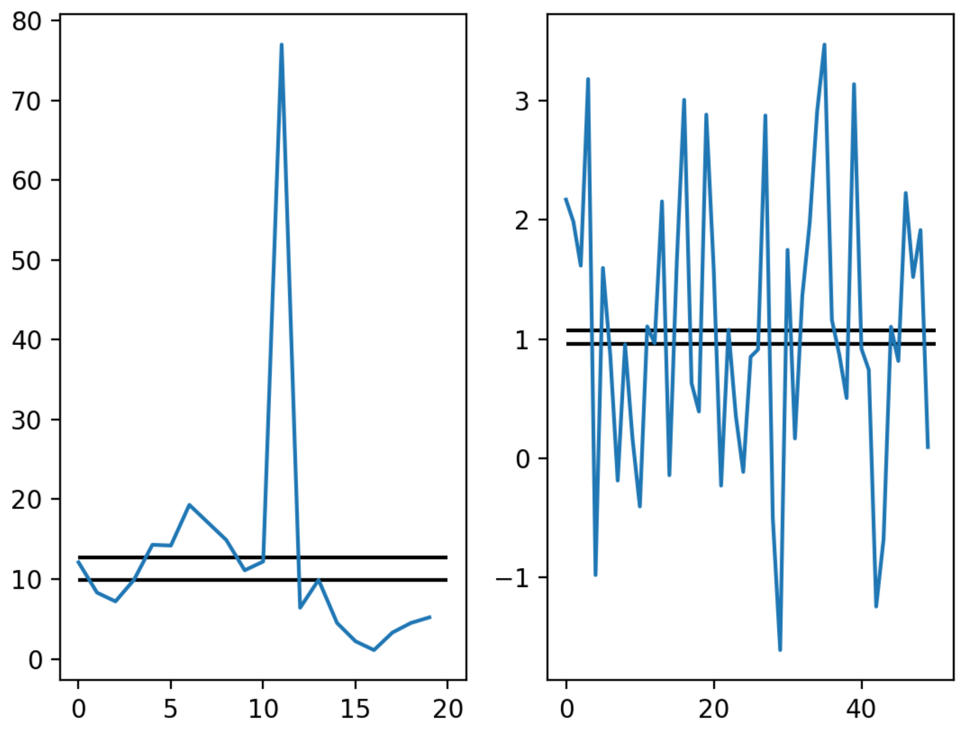
You will most likely not see a very smooth curve. This is because there are only 50 points in the x-axis array, which is not enough to make it look smooth. In math, our x-axis represents the real numbers, and there are an infinite amount of them - but our computers can't store an infinite amount of data! So we have to choose how many points in our x-axis to have by using a different step-size within the same range of 0 to 50.

Try to create the same x-axis array with 50 points, but this time with a step size of 0.1, which would increase the number of x-values to be 10 times as much with a step size of 1 (but still have a range from 0 to 50). Have a look at how the sine function looks now. It should look much smoother. Experiment with different step sizes to see the effect on the sine graph (try step sizes more than 1 as well).

4 Reading data from a file and plotting

In lecture, we saw how to read data line-by-line, which can be useful depending on how the file is structured. In this section, you are going to read data from files that are structured in different ways, and you will perform and plot some basic statistical calculations on that data.

1. Create a file named `lab8_part4.py`. Import both `numpy` and `matplotlib.pyplot` in the **same way** as we did in the previous files.
2. Download `data1.txt` and `data2.txt` from the Labs page and place them in the same folder as your Python file.
   Have a look at these two files. Notice that one of them has data separated horizontally by spaces, and the other one is separated by newlines.
3. The `np.loadtxt` method will read a file with data that has some sort of separation character, and them read the data in that file into a numpy array.
   `np.loadtxt(filename)` takes the name of the file you want to read.
   Use it to load the data in `data1.txt`, and save the result to a variable called `data`. Print `data` to the shell. What is the value of the variable? What is the variable's type?
4. Use the `plt.plot` function to plot the data points. Try plotting the data variable without giving an x-axis array - it will by default create one that is the length of `data`. Don't forget to call `plt.show()`.
5. Find the mean (average) of the data. Use the `np.mean()` function with the `data` array. Assign the mean value to a variable called `data_mean` in the Python file.
6. Find the median of the data. Using the `np.median()` function with the `data` array as an argument. Assign the median value to a variable called `data_median` in the Python file.
7. Now, you're going to add horizontal lines onto the plot that that indicate the mean and median of the values.
   Straight lines can be plotted using the `plt.hlines()` function.
   The code `plt.hlines(data_mean, 0, len(data))` will draw a solid horizontal line across the graph at the mean of the data.
   Write another command to plot a horizontal line at the median.
8. Run your code and check if your plot is correct. Make sure `plt.show()` is the last line in your file.
9. You will now convert your plotting script into a function to use with any data file.
   Define a function called `plot_with_mean_and_median(filename: str) -> None`
   which takes as an argument a file name. This function will load data from the file, plot the data, as well as the lines for the mean and median.
   After defining your function, comment out any commands outside of the function.
10. You will now plot two different graphs on the same plot, but on **two different figures**.
    Outside of the function, set up a **subplot** by first using the command `plt.subplot(1, 2, 1)`. The numbers indicate the number of rows, columns, and the order of the current graph on the plot.
    Now call your `plot_with_mean_and_median` function on an array made from `data1.txt`. Do not run your Python file just yet.
11. **After** the last function call, set up another **subplot** by first using the command
    `plt.subplot(1, 2, 2)`.
    Now call your `plot_with_mean_and_median` function on an array made from `data2.txt`.
12. Now, with `plt.show()` at the end of the code, run your Python file. On the next page is a sample of what you should see.



If you are having trouble making the above graphs, ask your TA for help.

Lastly, save the figure to an image file called `lab8_part4.png`. Check to make sure the file was saved to your folder.

5 Installing `matplotlib` on your computer

Follow these instructions to install `matplotlib` on your computer (choose the right instructions for either Mac or Windows). `numpy` comes with matplotlib so you don't need to install it seperately.

Mac: Open the Terminal application (spotlight search for 'Terminal').
Type the command:

```
python3.7 -m pip install matplotlib
```

Press Enter.

Windows: Open the Command Prompt application (Start menu search for 'Command Prompt' or 'cmd').
Type the command:

```
py -3.7 -m pip install matplotlib
```

Press Enter.

If you still cannot install it, talk to your TA or the instructor in lab, office hours, or after lecture.