Session 4: Python Tutorial - Modules and Libraries

Prepared by: Afiq Azraei, Harith Zulfaizal

December 11, 2020

1 Introduction

As your script grows in length and also complexity, it can be tedious when you need to find a particular line of code or make changes to it. Previously, we've learnt about functions and how they are useful to compartmentalize a few lines that are repeatedly used in different instances. Though it is not long until functions alone will not be enough to stop the script from becoming more complex. You will waste more time trying to find the lines than making the wanted changes.

Python, and other programming languages as well, has a feature that enables you to split a script into multiple files. This compartmentalization makes it easier to debug and to sift through the lines of code when needed. A side note on compartmentalization, it comes with its own benefit despite the its fractured nature:

- Easier long-term maintenance of scripts
- Less time spent explaining scripts to colleagues
- Easier re-usability of programmed functionality
- Faster short-term editability and code management

2 Modules

In general terms, a module is a piece of software that has a specific functionality. Just like how organizations are commonly subdivided into bureaus that handle different tasks, a program can be composed of a main script with supporting modules that help or adds to its function. Directly, modules are files that contain lines of code that can be imported into a script to be used later.

Python modules are basically '.py' files, with the name of the file acting as the name of the module. To use them in another script, one just imports them in using the keyword `import`. You can also use other modules in a module. In the example below, I created a module with some mathematical operations. This file will be named `CircleMaths.py`

```
pi = 3.14

def area(rad):
    return pi*(rad)**2
```

```
def circumference(rad):
    return 2*pi*rad
```

Later, we will import `CircleMaths.py` into the main script and use it there. To use an object from the module, we will use the . operator.

```
import CircleMaths

rad = 5

print(CircleMaths.area(rad))
```

2.1 Importing Modules

There are a few ways to import a module into a script, as shown below. It is common for modules to be imported early in the script, but not necessary. Sometimes, you may import modules later due to ease of tracking from where a function is imported.

```
# importing the module by name
import module
module.function()

# importing the module by another name
import module as mod
mod.function()

# importing a function from the module
from module import function1, function2
function2()

# importing everything from a module
from module import *
function1()
```

The last method is not encouraged, as you can easily be confused from where a function is originating from. You are also introducing a new set of names, which can override previously defined or declared ones.

Exercise Write modules for calculating the surface areas and volumes for common 3-D shapes, and area and perimeter for 2-D shapes. Calculate the values for a pyramid and a cylinder.

2.2 Module Path

When you import a module, Python goes through the following in order:

- The top level directory, where the current script is run
- PYTHONPATH directories
- Installation paths in Linux/Unix

You can easily find out where a module is located by checking the `__file__` variable:

```
import module
print(module.__file__)
```

Using `dir()`, you can also check all the variables and functions defined in a module.

3 Standard Modules

Python has a few libraries that come along with it, and most of them are really useful to know.

3.1 sys - System Specific Parameters and Functions

This is a module for variables that are used by the interpreter and functions that enable you to interact with them. The commonly used ones are:

```
import sys

# when you execute a script followed by arguments
# e.g.: 'python matrix.py 3' where 3 is an argument,
# you can access them by
print(sys.argv[0]) # this will print out the script's name
print(sys.argv[1]) # this will print out the following arguments and so on

# to immediately exit from a python, this is one of the methods used
sys.exit(0)
```

3.2 os - Miscellaneous OS Interfaces

As the name says, its a module to interact with your OS. Commonly utilised functions are:

```
import os

# creating a directory/folder
os.mkdir(path)

# deleting a file, not a directory
os.remove(path)

# deleting a directory
os.rmdir(path)

# another way to exit the current process
os._exit(0)

# executing a command in a subshell
os.system(cmd)
```

3.3 math - Mathematical Functions

Later, we will go through NumPy but sometimes a simple operation can be done with just `math` too.

```
import math

# rounds up a float
math.ceil(x)

# calculates factorial
math.factorial(x)

# calculates e to the power of x
math.exp(x)

# you also have trigo functions in there
math.sin(x)
math.atan(x)
```

3.4 random - Pseudo-random Number Generator

It basically generates number, in an almost random nature. To generate, it needs a seed first to initialize.

```
import random as rng

# generate a random integer between a and b
rng.randint(a, b)

# generare a random float between a and b
rng.uniform(a, b)
```

Exercise A hot object cools down faster than a warm object. The rate of heat outflow therefore decreases as an object cools. Using Newton's Law of Cooling (below) and the `math` and `random` modules, with the environmental temperature at 50K, initial temperature of object 300K and r has a value of 0.5, try to obtain a time when the temperature is between 120-130K.

$$T_t = T_{env} + (T_0 - T_{env})e^{-rt} \tag{1}$$

4 Packages

As of now, you might be accustomed already to the concept of modules. Now, we will go through one more level which is packages. They are essentially a collection of modules in the same directory, where the directory must have an empty file for initialization, named `__init__.py` that tells Python that this directory is a package. Therefore, any .py file inside can be imported. An example of such a directory would look like:

```
utils/
|-- __init__.py
|-- constants.py
|-- models.py
|-- DataViz/
|    |--__init__.py
|    |--Plotters.py
|    |--Aesthetics.py
```

In the example above, we made a package called `utils` that contain utility variables and functions that you may use a lot. The package has the initialization file, 2 modules and also a subpackage that has 2 more modules. To import them, it is as easy as:

```
import utils.constants
import utils.models
import utils.DataViz.Plotters
```

5 NumPy: Better Arrays

Python's implementation of array as a data structure is simplistic, but comes with its own drawbacks. Here comes `NumPy` to try to improve on it.

An open source library, it is one of the most widely used libraries among Python users especially for the STEM community. It is fairly easy to learn and understand, and it is integrated seamlessly in multiple other libraries which makes it much more appealing.

Why? NumPy arrays have a few tweaks compared to Python `lists` which optimizes it for fast computing of large sets of numbers. NumPy arrays are fixed size and store only a single data type, unlike their dynamic counterparts. They are also faster due to vectorization, i.e. non-explicit array operations e.g. indexing, looping etc.

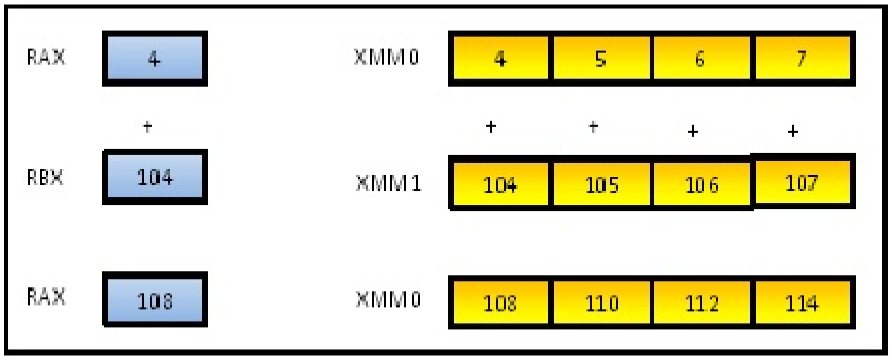


Figure 1: An example of vectorization

5.1 Basics

Different from the built-in `list` class, NumPy implements its own `ndarray`, which is a homogenous multidimensional array. An example of using NumPy is:

```
import numpy as np

arr1 = np.arange(10)
print(arr1)

arr2 = arr1.reshape(2, 5)
print(arr2)
```

To create arrays using NumPy, there are several methods.

- You can either use the `array()` method to cast `list`s or tuples into `ndarray`
- Use `zeros()` or `ones()` to create arrays of zeroes or ones respectively
- `empty()` creates an array with random float numbers
- `arange` is like the built-in `range()` except that it actually returns an array and not an iterator object

There are a few terms that may differ for `ndarray`. It's dimensions are called **axes** (plural of axis). A 2D array therefore has 2 axes. The **shape** of an array corresponds to the length of the arrays along an axis.

```
# casting a list using array()
a = [5, 4, 2, 1]
arr1 = numpy.array(a)
print(arr1)

# create an array full of 1
arr2 = numpy.ones(2, 2)
print(arr2)

# create an empty array
arr3 = np.empty(2, 3, 3)
print(arr3)

# create an array using arange()
arr4 = np.arange(1, 2, 0.2)
print(arr4)
```

These arrays can also be sliced, their members can accessed via indexing and they can also be iterated like normal arrays. Though for multidimensional arrays, indexing is done via tuples instead.

```
def f(x, y):
    return x**2 + y

arr = np.fromfunction(f, (3, 3))
print(b)
print(b[1,1])
```

Since arrays are essentially NumPy objects, they have attributes that can be accessed too. Some of the attributes of arrays that might be useful are:

- `ndim` - number of dimensions of the array
- `size` - the number of elements in the array
- `shape` - returns a tuple that indicates the number of elements along each dimension

5.2 Arithmetics

If you notice before, the `+` operator for `list` is used to concatenate them but for NumPy arrays they can actually add them element-wise, such that the first element in each array is summed together and so on. This applies to other arithmetic operators as well.

```
a = np.arange(1, 10)
b = np.arange(11, 20)

# adding two NumPy arrays will carry out the addition per element
print(a+b)
print(a-b)

# arithmetic operations directly on an array is carried out on each element
print(a**2)
print(np.cos(a))
print((a%2)==0)

# to perform dot product, use @ or .dot()
print(a.dot(b))
print(a@b)
```

NumPy has also other mathematical functions that are available in the `math` library such as sin, cos and exp. As you have also seen in the example code above, the `+` operator does not work the same for arrays and lists. For lists, it concatenates while for arrays it adds in a vectorized way. To concatenate arrays, you then use `np.concatenate(arr1, arr2)` instead.

Arrays can also be sorted using `np.sort(arr)` which automatically sorts the array in anscending order.

5.3 Shape Manipulation

Each NumPy array has an attribute called `shape` that returns its dimensions. Their shapes can also be changed via certain commands.

```
arr = np.arange(10).reshape(2,5)
print(arr)

# modifies the shape
print(arr.reshape(5,2))

# transposes the array
print(arr.T)

# stacking arrays
arr2 = np.arange(10).reshape(2,5)
arr3 = np.vstack((arr,arr2))
arr4 = np.hstack((arr,arr2))

print(arr3)
print(arr4)

#splitting arrays
print(np.hsplit(arr4, 2))
print(np.vsplit(arr3, 2))
```

Previously, we showed how you can access an element or slice an array. That slicing can also be done in a multidimensional way such that it returns an array with the same number of axes but not neccesarily the same shape.

```
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
arr1 = arr[0:2, 1:3]
```

In the example above, we declared a 2D array with a (2,5) shape. We then sliced the array to create a new one. You first specify the range on the first axes, and then the range on the second axes, etc.

Exercise Use a random number generator and create a (3,3) array. Calculate the determinant of the matrix manually, and compare your value with `np.linalg.det(arr)`.

6 Matplotlib: Data Visualisation

Matplotlib is a data visualisation library that is widely used by many people, due to its simplicity and also integration with other libraries. It is possible to produce simple plots very quickly, and also high quality, publication worthy plots too.

A simple plot can be done fairly easily:

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

x = np.arange(10)
y = x**2

plt.plot(x, y, label='quadratic')
plt.xlabel('x')
plt.ylabel('y')
plt.title('y = x^2')
plt.legend()
plt.show()
```

There are generally two ways to plot using Matplotlib, one being the MATLAB style as shown before and the other with an object-oriented style. We will stick with the first most of the time, as you may not be accustomed yet to the latter.

6.1 Plot Style Formatting

Unless specified, Matplotlib will default to the `b-` where `b` stands for the color blue and the dash tells Matplotlib to plot a solid line connecting the points. You can also plot with other styles by specifying it in the third argument. Below are examples of other formatting styles, and you can mix them up according to your creativity.

Character	Description
g.	Green points
ro	Red circles
bˆ	Blue upright triangles
y+	Yellow plus markers
k–	Black dashed line
m:	Magenta dotted line
cs	Cyan square marker

Asides from Cartesian plots, Matplotlib also has a set of other plot styles too. There are scatter plots, bar plots, and historgram plots, each using the `scatter()`, `bar()` and `hist()` methods from Matplotlib.

If you want to make the plot logarithmic, just change the scaling via `plt.xscale('log')` or `plt.yscale('log')`

To change the range of the axis, we use `plt.axis([0, 10, 0, 10])` for example where the first two arguments are the range for the x-axis and the latter two for the y-axis.

Aside from the stuff we mentioned, others are up to you to explore using the Matplotlib documentation.

Exercise The Bisection Method is one of the many root-finding methods for continuous functions. Using the equation given below, find its root and plot the function and its root using the libraries you have learnt.

$$f(x) = x^3 - x - 2 \tag{2}$$

7 Pandas: Data Manipulation & Analysis

Pandas is an open-source library for high-level data manipulation and analysis using its various data structures. The name *Pandas* is derived from the word *panel data* usually found and used in statistics and econometrics. It is built on NumPy package and its data structures consists of Series, DataFrame and Panel. In this section, we are going to be focused on Series and DataFrame, two of the most popular and widely used data structures.

Generally, a **Series** is a one-dimensional array with built-in indexing that is capable of holding any data types while a **DataFrame** is a two-dimensional array of the same properties in rows and columns.

```
import pandas as pd

a = pd.Series() #empty Series
print(a)

lst = [1, 2, 4, 6, 9, 12, 15, 26, 31]
b = pd.Series(lst)
print(b)

person = ['Ali', 'Abu', 'Karl', 'Rosa', 'Sarah', 'Nurul', 'John']
age = [12, 32, 64, 56, 43, 24, 19]

empty_df = pd.DataFrame() #empty DataFrame
print(empty_df)

df = pd.DataFrame({'Name': person, 'Age': age})

df1 = pd.DataFrame(list(zip(person, age)), columns=['Name', 'Age'], index
    =[1,2,3,4,5,6,7]) #if indices are not defined, the DataFrame will be zero
    indexed by default
print(df1)
```

As seen in the code block above, we are able to define empty Series and DataFrame, and also Series and DataFrame with data embedded. There are several ways to create a DataFrame, either by defining columns one by one `df['Name'] = person` or by defining them during the creation of the DataFrame, `pd.DataFrame(data, columns, index)`.

7.1 Basic Methods

The Series and DataFrame comes with some basic functionality that might be useful in some cases such as `axes, ndim, size, values, shape, T, head, tail` which are pretty obvious. These functionalities or methods are called using the syntax `<your dataframe>.<method>`. Try them out and observe the outputs of each method:

```
print('Axes:', b.axes, 'Dimension:', b.ndim)
print('Size:', b.size, 'Values:', b.values)

print('Axes:', df.axes, 'Dimension:', df.ndim, 'Size:', df.size)
print('Shape:', df.shape)
print('Values:', df.values)
print(df.T) #transpose
print(df.head(2))
print(df.tail(2))
```

7.2 Descriptive Statistics

Apart from the basic methods, there are a lot of other methods on descriptive statistics that can be operated and used on a DataFrame. Some of the famous methods include `sum, count, mean, std` among others.

```
print('Sum:', df.sum(), 'Mean:', df.mean(), 'Std:', df.std())
print('Sum_AgeOnly:' df['Age'].sum()) # Calling columns by their names

print(df.describe()) #gives a summary of statistics
```

7.3 Rows and Columns

With a Pandas DataFrame, it is possible to select only certain column(s), adding entirely new column(s) to existing DataFrame and even deleting column(s).

```
print(df['Name'])

height = [104, 172, 181, 154, 156, 163, 170]
birthplace = ['Kepala Batas', 'Seremban', 'Georgetown', 'Kangar', 'Tawau', '
    Kota Tinggi', 'Seputeh']

df['Height'] = height
df['Birthplace'] = birthplace

print(df)
print(df.drop(['Age'], axis=1, inplace=True))
```

It is also possible to do these on rows. For calling up row values/elements of a DataFrame, methods `.loc[]` and `.iloc[]` are used. The former is used if you want to call up a row by its name, while the latter uses the index of the rows.

```
print(df.loc[2:5])
print(df.iloc[2:5])

df.set_index('Name', inplace=True)
print(df)
print(df.loc['Karl'])
```

The method `.set_index(column, inplace=True)` is used to change the default index to the names as the index. Adding or concatenating new rows into existing DataFrame can be done by using the method `.concat([new_df, old_df])`.

```
new_df = pd.DataFrame({'Name': 'Mustafa', 'Height':181, 'Birthplace': 'Bukit
    Jelutong'}, index=[0])

df = pd.concat([new_df,df]).reset_index(drop=True)
print(df)

print(df.drop([1,2], axis=0)) #drop rows
```

`.reset_index(drop=True)` is needed due to the change of index we did earlier. If the DataFrame already has the default index, the method is not needed. The `.drop()` can be used on the row's label or the index of the row as shown above.

7.4 Plotting

Columns from a DataFrame can be plotted using Matplotlib per usual, `plt.plot(x,y)` or by applying a method to the DataFrame, `df.plot(x,y)`. For example,

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

plt.rcParams["font.family"] = "serif"

dist = [1,2,3,4,6,8,10,12]
curr = [6.3,8.3,10.6,13.7,22.8,38.8,69,130]
curr_ln = [np.log(x*10**12) for x in curr]

ts_discharge = pd.DataFrame(list(zip(dist, curr_ln)), columns=['Distance', '
    Current'])

ts_discharge.plot(x='Distance', y='Current') #method .plot
plt.xlabel('Inter-electrode distance, d (mm)')
plt.ylabel('ln I')
plt.show()

plt.plot(ts_discharge['Distance'], ts_discharge['Current']) #plt.plot
plt.show()
```

Besides the simple x and y scatter and line plot, Pandas comes with a vast library of built-in data visualisation tools that is based on Matplotlib such as `df.plot.area, df.plot.barh, df.plot.hist, df.plot.density, df.plot.box, df.plot.pie` among others.

7.5 Importing CSV

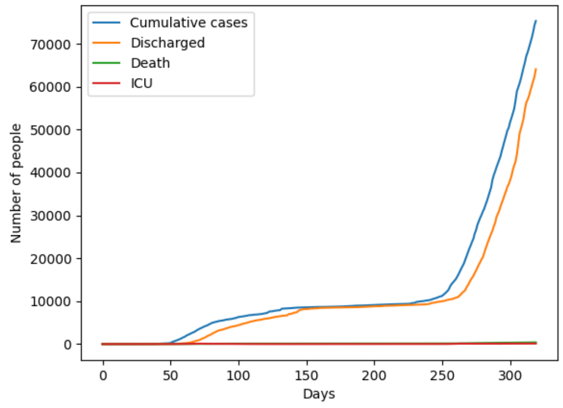
In Pandas, it is possible to import csv and xlsx files into DataFrame(s). The process of loading up data from csv and xlsx file(s) can be achieved by using the function `pd.read_csv()` or `pd.read_excel()`. It also possible to save your DataFrame into a csv or xlsx file via `.to_csv()` and `.to_excel()`.

```
#download csv file from
#https://github.com/ynshung/covid-19-malaysia/blob/master/covid-19-my-states-
    cases.csv

import pandas as pd

df = pd.read_csv("..\covid-19-my-states-cases.csv")
print(df)

df.to_csv(output_path)
```

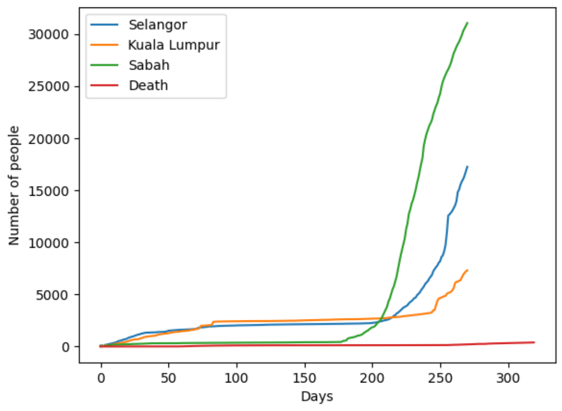
By default, csv as its name implies is separated by comma (,) and the function `pd.read_csv()` defines the separator, `sep` as comma (,). If the csv file you are handling has a different separator, you should identify the separator in the function. Eg: `pd.read_csv(file, sep=":")` for colon separator or the built-in `pd.read_csv(file, delim_whitespace=True)` for whitespace delimiter.

Exercise By using the data from *https://github.com/ynshung/covid-19-malaysia* of both covid-19-malaysia.csv and covid-19-my-states-cases.csv.

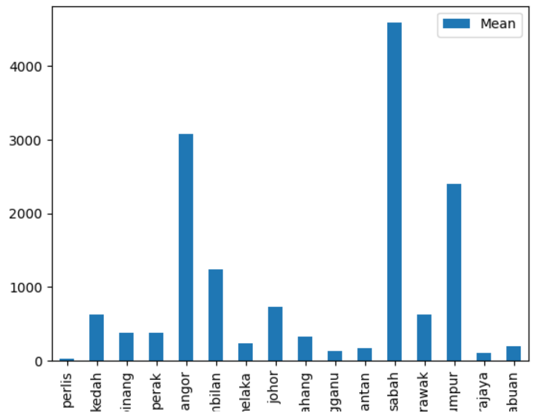
1. Plot the graph of cumulative cases, discharged, death and ICU from Covid-19 for the country as a whole against days.



2. Plot the graph of cumulative cases of Selangor, Kuala Lumpur and Sabah with the cumulative death toll against days.



3. Plot a barchart of the means of cases of every states.


Mean
perlis
kedah
inang
perak
angor
nbilan
elaka
johor
ahang
gganu
antan
sabah
rawak
umpur
rajaya
abuan