PROGRAM STUDI INFORMATIKA
FAKULTAS TEKNIK DAN INFORMATIKA
UNIVERSITAS MULTIMEDIA NUSANTARA
SEMESTER GENAP TAHUN AJARAN 2021/2022

IF420 – ANALISIS NUMERIK

Pertemuan ke 3 – More on Python

Seng Hansun, S.Si., M.Cs.

Capaian Pembelajaran Mingguan Mata Kuliah (Sub-CPMK):

Sub-CPMK 3: Mahasiswa mampu menerapkan dan memanfaatkan beberapa fitur lanjutan di Python – C3

Reviews

- Recursion
- Object Oriented Programming (OOP)
- Complexity
- Representation of Numbers
- Errors, Good Programming Practices, and Debugging

Outlines

- Reading and Writing Data
- Visualization and Plotting
- Parallel Your Python

Reading and Writing Data

- **Storing** data and the results of your programming efforts is important for working over multiple sessions and sharing your results with collaborators.
- Since when Python closes, all the variables in the memory are **lost**, data must be **stored** in some other way.
- Sometimes data must also be **readable** by or **written** in a form that can be read by other programs.
- We will discuss several means in storing data, such as **TXT** files, **CSV** files, **Pickle** files, **JSON** files, and **HDF5** files.

TXT Files

- A **text file**, many times with an extension **.txt**, is a file containing only **plain text**. However, programs you write and programs that read your text file will usually expect the text file to be in a certain format; that is, organized in a specific way.
- To work with text files, we need to use **open** function which returns a **file object**. It is commonly used with two arguments.

```
f = open(filename, mode)
```

- Example: Create a text file called **test.txt** and write a couple lines in it.

In [1]:
```
f = open('test.txt', 'w')
for i in range(5):
    f.write(f"This is line {i}\n")

f.close()
```

test - Notepad

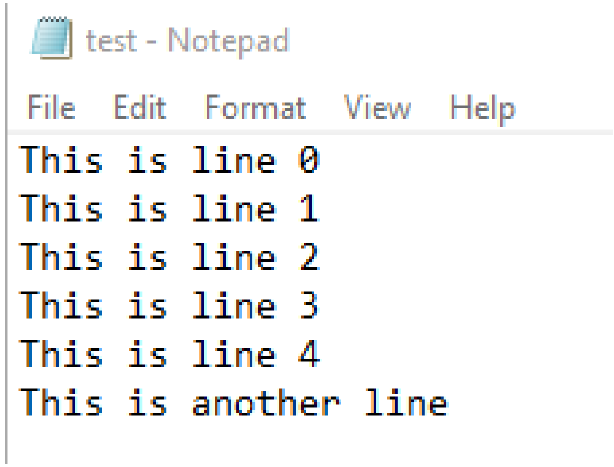
File Edit Format View Help

```
This is line 0
This is line 1
This is line 2
This is line 3
This is line 4
```

TXT Files

- Now, let's **append** some string to the test.txt file. It is very similar to how we write the file, with only one difference - change the mode to '**a**' instead.

```
In [2]:  f = open('test.txt', 'a')
         f.write(f"This is another line\n")
         f.close()
```



TXT Files

- We could **read** a file from disk and store all the contents to a **variable**.

```
In [3]:  f = open('./test.txt', 'r')
         content = f.read()
         f.close()
         print(content)

         This is line 0
         This is line 1
         This is line 2
         This is line 3
         This is line 4
         This is another line
```

```
In [4]:  type(content)

Out[4]:  str
```

- But sometimes we want to read in the contents in the files **line by line** and store it in a **list**. We could use **f.readlines()** to achieve this.

```
In [5]:  f = open('./test.txt', 'r')
         contents = f.readlines()
         f.close()
         print(contents)

         ['This is line 0\n', 'This is line 1\n', 'This is line 2\n', 'This is line
         3\n', 'This is line 4\n', 'This is another line\n']
```

```
In [6]:  type(contents)

Out[6]:  list
```

Dealing with numbers and arrays

- Since we are working with **numerical methods** later, and many times, we work with the **numbers** or **arrays**. We could use the above methods to save the numbers or arrays to a file and read it back to the memory. But it is **not so convenient** this way.
- Instead, commonly we use the **numpy** package to directly **save/read** an array.

```
In [7]:  import numpy as np

In [8]:  arr = np.array([[1.20, 2.20, 3.00], [4.14, 5.65, 6.42]])
         arr

Out[8]: array([[1.2 , 2.2 , 3.  ],
               [4.14, 5.65, 6.42]])

In [9]:  np.savetxt('my_arr.txt', arr, fmt='%.2f', header = 'Col1 Col2 Col3')
```

my_arr - Notepad

File Edit Format View Help

```
# Col1 Col2 Col3
1.20 2.20 3.00
4.14 5.65 6.42
```

```
In [10]:  my_arr = np.loadtxt('my_arr.txt')
          my_arr

Out[10]: array([[1.2 , 2.2 , 3.  ],
                [4.14, 5.65, 6.42]])
```

- We can see, read in the file directly to an array is very simple using the np.loadtxt function and it **skips** the **first header** as well.

CSV Files

- There are many scientific data which are stored in the **comma-separated values** (CSV) file format, a **delimited text** file that uses a **comma** to separate values.
- It is a very useful format that can store large tables of data (numbers and text) in plain text.
- Each line (row) in the data is **one** data **record**, and each record consists of **one or more fields**, separated by **commas**.
- It also can be opened using Microsoft Excel, and visualize the rows and columns.

CSV Files

- We use the **np.savetxt** function to save the data to a **csv** file.

```
In [11]:  import numpy as np

          data = np.random.random((100,5))

          np.savetxt('test.csv', data, fmt = '%.2f', delimiter=',', header = 'c1, c2, c3, c4, c5')
```

	A	B	C
1	# c1, c2, c3, c4, c5		
2	0.94,0.71,0.55,1.00,0.51		
3	0.70,1.00,0.68,0.13,0.92		
4	0.89,0.00,0.33,0.28,0.72		
5	0.67,0.22,0.16,0.73,0.24		
6	0.78,0.49,0.91,0.43,0.43		
7	0.06,0.27,0.27,0.39,0.88		
8	0.66,0.65,0.38,0.45,0.69		
9	0.17,0.45,0.73,0.88,0.01		
10	0.12,0.22,0.59,0.38,0.59		
11	0.87,0.64,0.56,0.97,0.67		

- We could read in the csv file using the **np.loadtxt** function.

```
In [12]:  my_csv = np.loadtxt('./test.csv', delimiter=',')
          my_csv[:5, :]

Out[12]: array([[0.94, 0.71, 0.55, 1.  , 0.51],
                [0.7 , 1.  , 0.68, 0.13, 0.92],
                [0.89, 0.  , 0.33, 0.28, 0.72],
                [0.67, 0.22, 0.16, 0.73, 0.24],
                [0.78, 0.49, 0.91, 0.43, 0.43]])
```

Pickle Files

- In this section, we introduce another way to store the data to the disk - **pickle**.
- We talked about saving data into **text file** or **csv file**. But in certain cases, we want to store **dictionaries**, **tuples**, **lists**, or any **other data type** to the disk and use them later or send them to some colleagues.
- Pickle can be used to **serialize** Python **object structures**, which refers to the process of converting an object in the memory to a **byte stream** that can be stored as a **binary file** on disk.
- When we load it back to a Python program, this binary file can be **de-serialized** back to a Python object.
- To use a **pickle**, we need to import the module first.

Pickle Files

- To use pickle to serialize an object, we use the **pickle.dump** function, which takes two arguments: the first one is the object, and the second argument is a file object returned by the **open** function.
- Note here the mode of the open function is ‘**wb**’ which indicates write binary file.

```
In [13]:  import pickle
```

```
In [14]:  dict_a = {'A':0, 'B':1, 'C':2}
          pickle.dump(dict_a, open('test.pkl', 'wb'))
```

- Now let’s load the pickle file we just saved on the disk back using the **pickle.load** function.

```
In [15]:  my_dict = pickle.load(open('./test.pkl', 'rb'))
          my_dict

Out[15]: {'A': 0, 'B': 1, 'C': 2}
```

JSON Files

- **JSON** stands for **JavaScript Object Notation**.
- A JSON file usually ends with extension "**.json**".
- Unlike pickle, which is Python dependent, JSON is a **language-independent** data format, which makes it attractive to use.
- Besides, it is usually takes **less space** on the disk and the manipulation is **faster** than pickle. Therefore, it is a good option to store your data using JSON.
- The text in JSON is done through **quoted string** containing value in key-value pairs within **{}**. It is actually very similar to the **dictionary** we saw in Python.

```python
In [16]: import json
```

```python
In [17]: school = {
    "school": "UC Berkeley",
    "address": {
        "city": "Berkeley",
        "state": "California",
        "postal": "94720"
    },

    "list":[
        "student 1",
        "student 2",
        "student 3"
        ],

    "array":[1, 2, 3]
}
json.dump(school, open('school.json', 'w'))
```

- In Python, the easiest way to handle JSON is to use **json library**. There are some other libraries such as **simplejson, jyson,** etc. Here, we will use **json** which is natively supported by Python to write and load JSON files.
- To use json to serialize an object, we use the **json.dump** function.
- To load the JSON file, we use **json.load** function.

```python
In [18]: my_school = json.load(open('./school.json', 'r'))
my_school

Out[18]: {'school': 'UC Berkeley',
 'address': {'city': 'Berkeley', 'state': 'California', 'postal': '94720'},
 'list': ['student 1', 'student 2', 'student 3'],
 'array': [1, 2, 3]}
```

HDF5 Files

- In scientific computing, sometimes, we need to store **large amounts** of data with **quick access**, the file formats we introduced before are not going to cut it.
- You will soon find there are many cases, **HDF5** (**Hierarchical Data Format**) is the solution. It is a **powerful binary data format** with **no limit on the file size**. It provides **parallel IO** (input/output), and carries out a bunch of **low level optimizations** under the hood to make the **queries faster** and **storage** requirements **smaller**.
- An HDF5 file saves two types of objects: **datasets**, which are array-like collections of data (like NumPy arrays), and **groups**, which are folder-like containers that hold datasets and other groups.
- The so called **hierarchical** in HDF5 refers to the fact that the data could be saved like a file system, with **folder-like structures**, such as folder, subfolder (in HDF5, it is called group, subgroup). Groups operate like dictionaries with the keys and values, with the **keys** are **names** of the groups, and the **values** are the **subgroups** or **datasets**.

HDF5 Files

- In order to use read/write HDF5 in Python, there are some packages or wrappers to serve the purposes. The most common two packages are **PyTables** and **h5py**.
- We will only introduce the **h5py** here. You can install h5py using **conda** (hope you still remember how to do that, if you forget, please go back to Chapter 1).
- **Example**: Suppose we deployed some instruments to monitor the accelerations and GPS location in Bay Area, CA. We deployed **two accelerometers** at **Berkeley** and **Oakland** as well as one GPS station at **San Fransisco**. And they record data at different sampling rates, with the accelerometer at Berkeley sample the data every **0.04 s**, and **0.01 s** for the sensor at Oakland. The GPS samples the location every **60 seconds** in San Fransisco. Now we want to store the two types of data into a HDF5 as well as some attributes indicate where the data is recorded, start time of the recording, station name and the sampling interval.

In [19]:
```python
import numpy as np
import h5py
```

In [20]:
```python
# Generate random data for recording
acc_1 = np.random.random(1000)
station_number_1 = '1'
# unix timestamp
start_time_1 = 1542000276
# time interval for recording
dt_1 = 0.04
location_1 = 'Berkeley'

acc_2 = np.random.random(500)
station_number_2 = '2'
start_time_2 = 1542000576
dt_2 = 0.01
location_2 = 'Oakland'
```

In [21]:
```python
hf = h5py.File('station.hdf5', 'w')
```

In [22]:
```python
hf['/acc/1/data'] = acc_1
hf['/acc/1/data'].attrs['dt'] = dt_1
hf['/acc/1/data'].attrs['start_time'] = start_time_1
hf['/acc/1/data'].attrs['location'] = location_1

hf['/acc/2/data'] = acc_2
hf['/acc/2/data'].attrs['dt'] = dt_2
hf['/acc/2/data'].attrs['start_time'] = start_time_2
hf['/acc/2/data'].attrs['location'] = location_2

hf['/gps/1/data'] = np.random.random(100)
hf['/gps/1/data'].attrs['dt'] = 60
hf['/gps/1/data'].attrs['start_time'] = 1542000000
hf['/gps/1/data'].attrs['location'] = 'San Francisco'
```

In [23]:
```python
hf.close()
```

Read in the HDF5 files

```
In [24]: hf_in = h5py.File('station.hdf5', 'r')

In [25]: list(hf_in.keys())
Out[25]: ['acc', 'gps']

In [26]: acc = hf_in['acc']

In [27]: list(acc.keys())
Out[27]: ['1', '2']

In [28]: data_1 = hf_in['acc/1/data']
```

- We can see reading a HDF5 is also easy with **h5py**. After we read in the HDF5 to **hf_in**, we could see what groups are in the HDF5 using the **keys** function.

```
In [29]: data_1[:10]
Out[29]: array([0.47186583, 0.47308128, 0.46795467, 0.01968574, 0.94769089,
                0.49005002, 0.7099151 , 0.76995257, 0.89032606, 0.23462921])

In [30]: list(data_1.attrs)
Out[30]: ['dt', 'location', 'start_time']

In [31]: data_1.attrs['dt']
Out[31]: 0.04

In [32]: data_1.attrs['location']
Out[32]: 'Berkeley'
```

- Then we could get access to the group members and see what contains in the subgroups as the **hf_in['acc']**, or directly specify the path to the datasets as **hf_in['acc/1/data']** and get the array data.

Visualization and Plotting

- **Visualizing data** is usually the best way to convey important engineering and science ideas and information, especially if the information is made up of many, many numbers.
- The ability to **visualize** and **plot data** quickly and in many different ways is one of Python's most **powerful features**.
- Python has **numerous graphics functions** that enable you to efficiently display plots, surfaces, volumes, vector fields, histograms, animations, and many other data plots.

2D Plotting

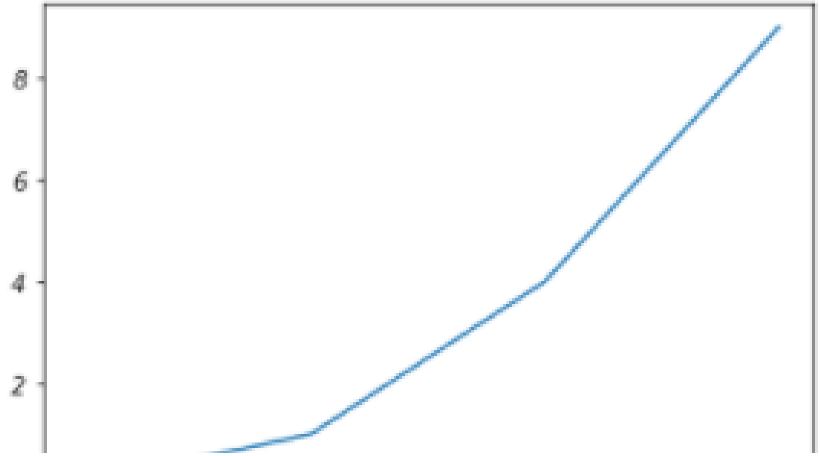
- In Python, the **matplotlib** is the most important package that to make a **plot**, you can have a look of the **matplotlib** gallery and get a sense of what could be done there.
- Usually the first thing we need to do to make a plot is to import the **matplotlib** package.
- In Jupyter notebook, we could show the figure directly within the notebook and also have the interactive operations like **pan**, **zoom in/out**, and so on using the magic command - **%matplotlib notebook**.

```
In [33]:  import numpy as np
          import matplotlib.pyplot as plt
          %matplotlib notebook
```

- The basic plotting function is **plot(x,y)**. The plot function takes in two lists/arrays, x and y, and produces a visual display of the respective points in x and y.

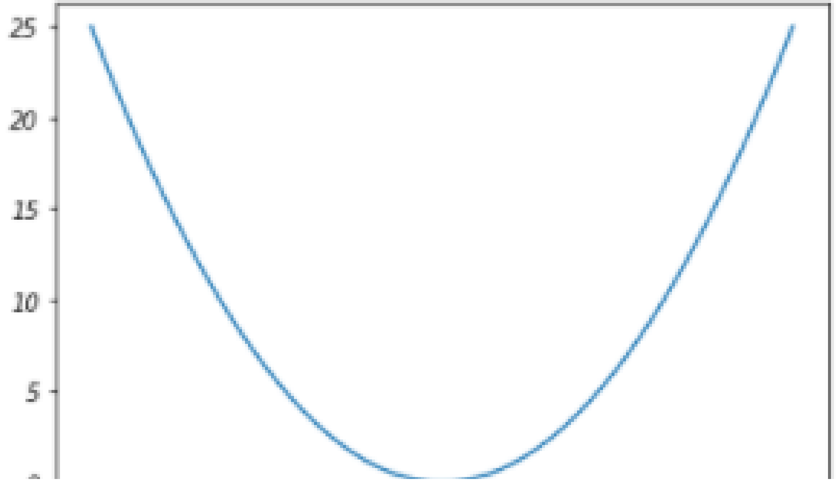
In [34]:

```python
x = [0, 1, 2, 3]
y = [0, 1, 4, 9]
plt.plot(x, y)
plt.show()
```



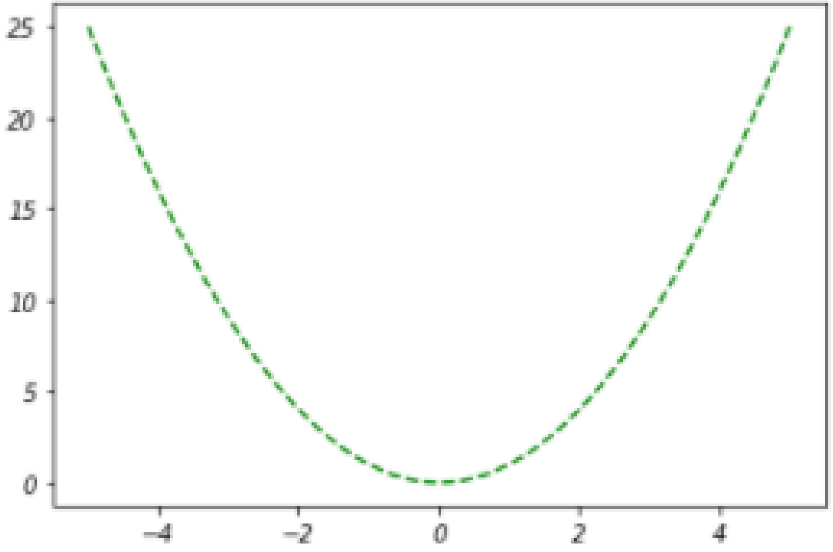
In [35]:

```python
x = np.linspace(-5,5, 100)
plt.plot(x, x**2)
plt.show()
```



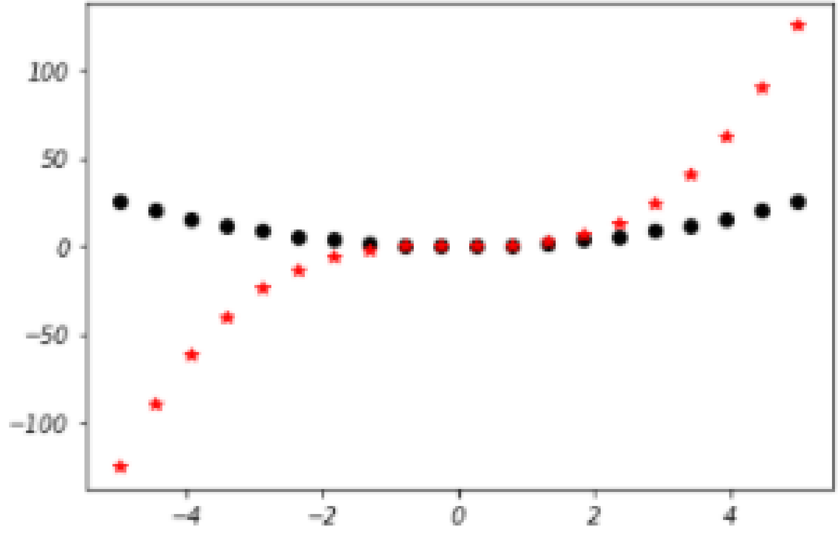
In [36]:

```python
x = np.linspace(-5,5, 100)
plt.plot(x, x**2, 'g--')
plt.show()
```

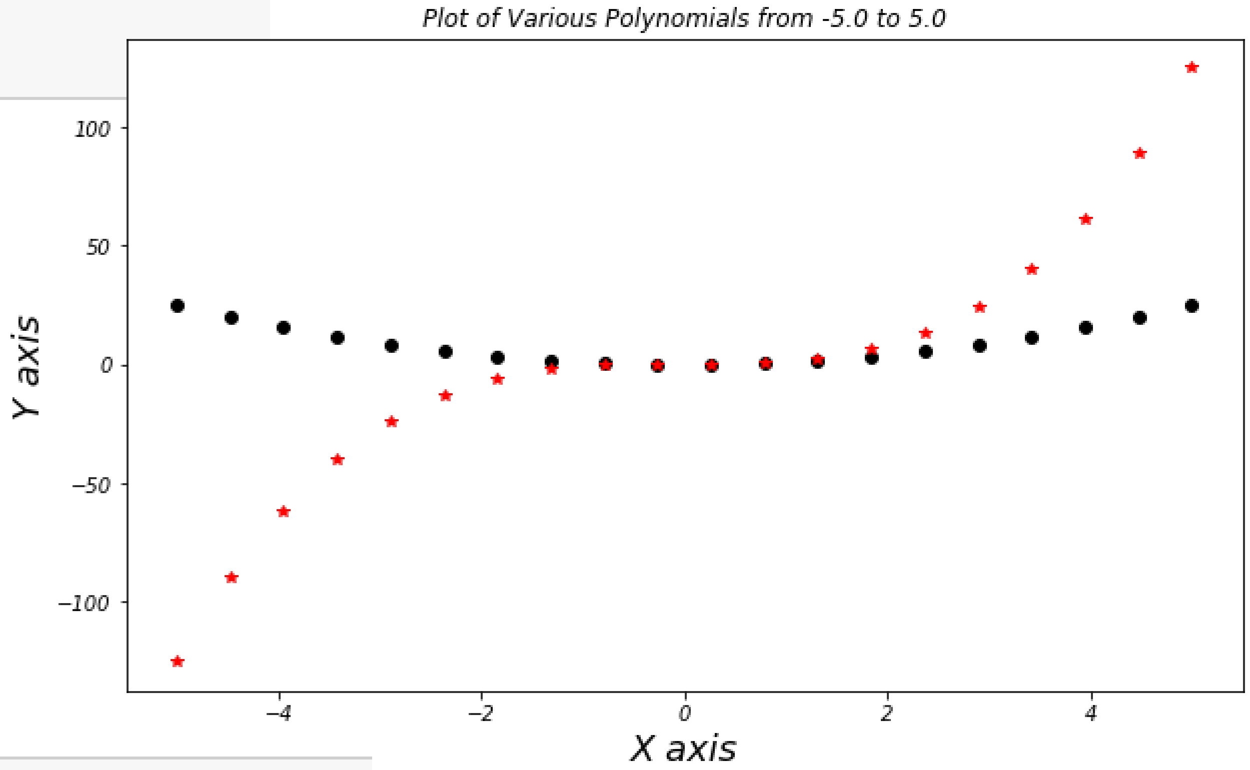


In [37]:

```python
x = np.linspace(-5,5,20)
plt.plot(x, x**2, 'ko')
plt.plot(x, x**3, 'r*')
plt.show()
```



In [38]:
```
plt.figure(figsize = (10,6))

x = np.linspace(-5,5,20)
plt.plot(x, x**2, 'ko')
plt.plot(x, x**3, 'r*')
plt.title(f'Plot of Various Polynomials from {x[0]} to {x[-1]}')
plt.xlabel('X axis', fontsize = 18)
plt.ylabel('Y axis', fontsize = 18)
plt.show()
```



- We can see that we could change any part of the figure, such as the x and y axis label size by specify a **fontsize** argument in the **plt.xlabel** function.
- But there are some **pre-defined** styles that we could use to automatically change the style. Here is the list of the styles.

In [39]:
```
print(plt.style.available)
```

```
['bmh', 'classic', 'dark_background', 'fast', 'fivethirtyeight', 'ggplot',
'grayscale', 'seaborn-bright', 'seaborn-colorblind', 'seaborn-dark-palett
e', 'seaborn-dark', 'seaborn-darkgrid', 'seaborn-deep', 'seaborn-muted', 's
eaborn-notebook', 'seaborn-paper', 'seaborn-pastel', 'seaborn-poster', 'sea
born-talk', 'seaborn-ticks', 'seaborn-white', 'seaborn-whitegrid', 'seabor
n', 'Solarize_Light2', 'tableau-colorblind10', '_classic_test']
```

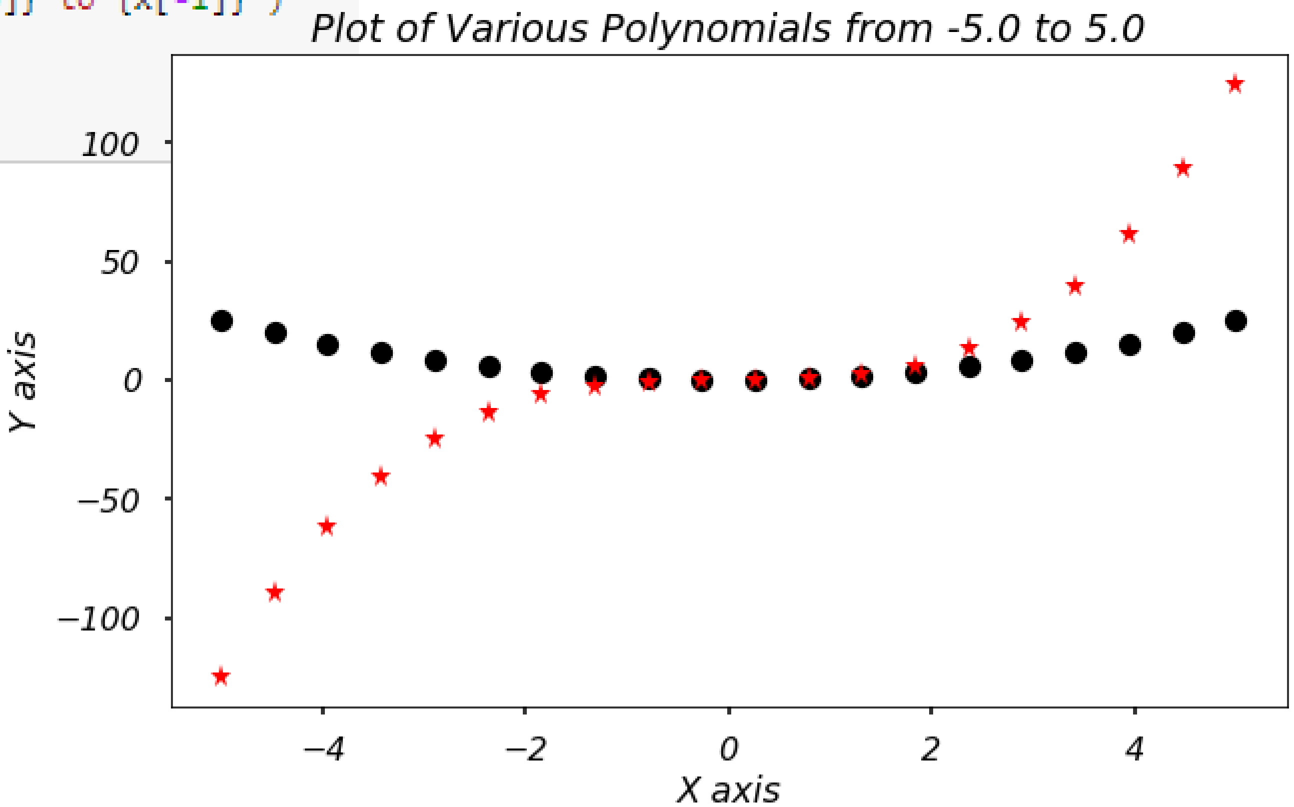
2D Plotting

In [40]:

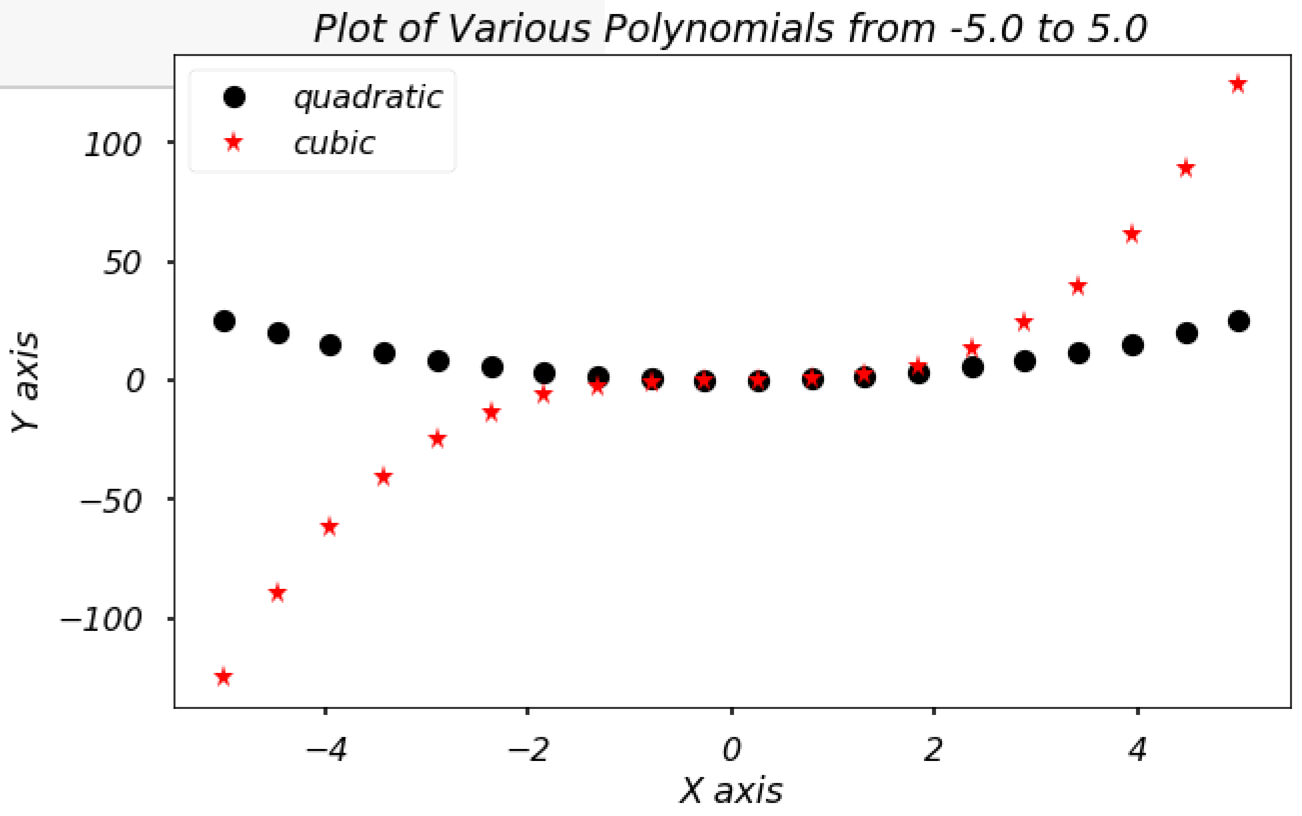
```
plt.style.use('seaborn-poster')
```

In [41]:

```
plt.figure(figsize = (10,6))

x = np.linspace(-5,5,20)
plt.plot(x, x**2, 'ko')
plt.plot(x, x**3, 'r*')
plt.title(f'Plot of Various Polynomials from {x[0]} to {x[-1]}')
plt.xlabel('X axis')
plt.ylabel('Y axis')
plt.show()
```



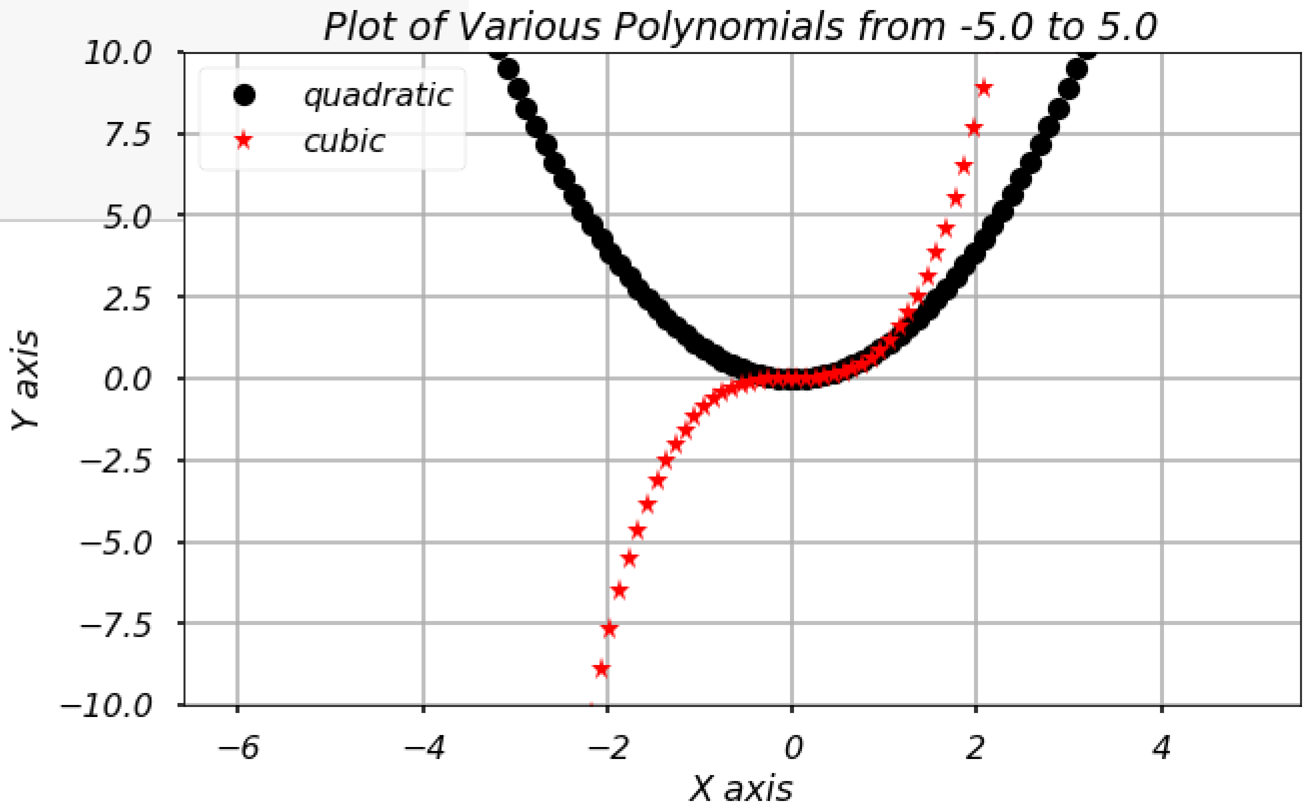
```
In [42]:
plt.figure(figsize = (10,6))

x = np.linspace(-5,5,20)
plt.plot(x, x**2, 'ko', label = 'quadratic')
plt.plot(x, x**3, 'r*', label = 'cubic')
plt.title(f'Plot of Various Polynomials from {x[0]} to {x[-1]}')
plt.xlabel('X axis')
plt.ylabel('Y axis')
plt.legend(loc = 2)
plt.show()
```



- You can add a legend to your plot by using the **legend** function and add a **label** argument in the plot function. The legend function also takes argument of **loc** to indicate where to put the legend, try to change it from 0 to 10.

In [43]:

```python
plt.figure(figsize = (10,6))

x = np.linspace(-5,5,100)
plt.plot(x, x**2, 'ko', label = 'quadratic')
plt.plot(x, x**3, 'r*', label = 'cubic')
plt.title(f'Plot of Various Polynomials from {x[0]} to {x[-1]}')
plt.xlabel('X axis')
plt.ylabel('Y axis')
plt.legend(loc = 2)
plt.xlim(-6.6)
plt.ylim(-10,10)
plt.grid()
plt.show()
```



- Finally, you can further customize the appearance of your plot to change the **limits** of each axis using the **xlim** or **ylim** function.
- Also, you can use the **grid** function to turn on the grid of the figure.

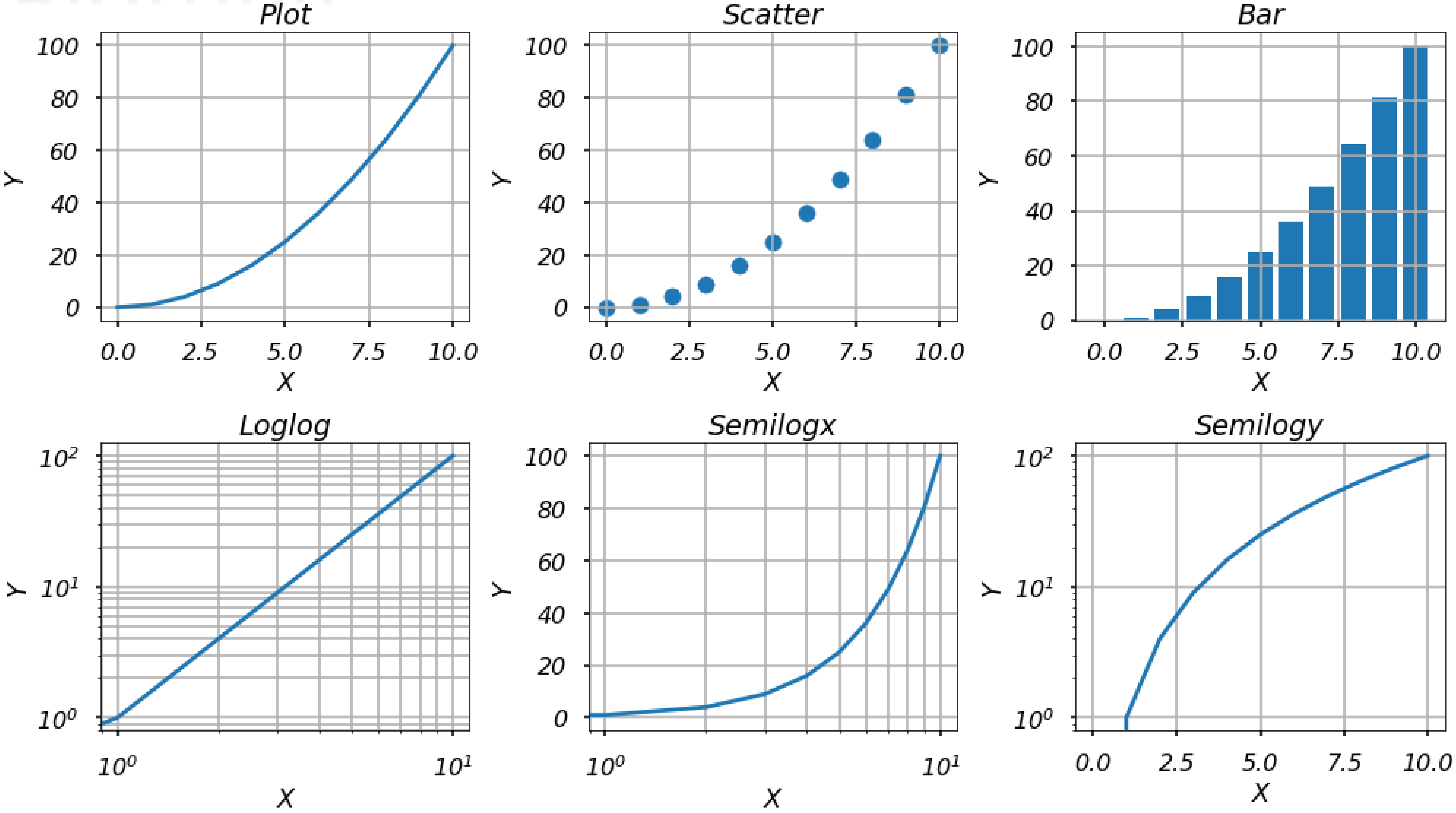
2D Plotting

- We can create a table of plots on a **single figure** using the **subplot** function.
- The subplot function takes **three inputs**: the number of **rows** of plots, the number of **columns** of plots, and to **which plot** all calls to plotting functions should plot.
- You can **move** to a different subplot by **calling** the **subplot** again with a different entry for the plot location.
- There are several other plotting functions that plot x versus y data. Some of them are **scatter**, **bar**, **loglog**, **semilogx**, and **semilogy**.

```python
In [44]: x = np.arange(11)
         y = x**2

         plt.figure(figsize = (14, 8))

         plt.subplot(2, 3, 1)
         plt.plot(x,y)
         plt.title('Plot')
         plt.xlabel('X')
         plt.ylabel('Y')
         plt.grid()

         plt.subplot(2, 3, 2)
         plt.scatter(x,y)
         plt.title('Scatter')
         plt.xlabel('X')
         plt.ylabel('Y')
         plt.grid()

         plt.subplot(2, 3, 3)
         plt.bar(x,y)
         plt.title('Bar')
         plt.xlabel('X')
         plt.ylabel('Y')
         plt.grid()
```

```python
plt.subplot(2, 3, 4)
plt.loglog(x,y)
plt.title('Loglog')
plt.xlabel('X')
plt.ylabel('Y')
plt.grid(which='both')

plt.subplot(2, 3, 5)
plt.semilogx(x,y)
plt.title('Semilogx')
plt.xlabel('X')
plt.ylabel('Y')
plt.grid(which='both')

plt.subplot(2, 3, 6)
plt.semilogy(x,y)
plt.title('Semilogy')
plt.xlabel('X')
plt.ylabel('Y')
plt.grid()

plt.tight_layout()

plt.show()
```

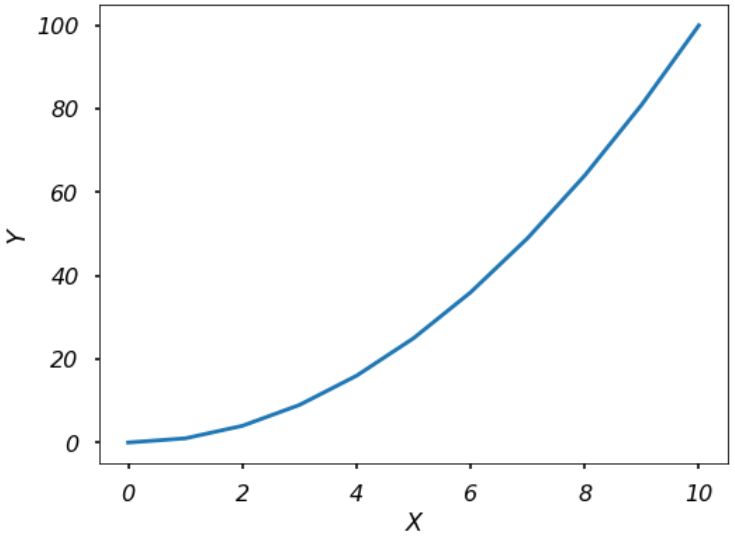
2D Plotting

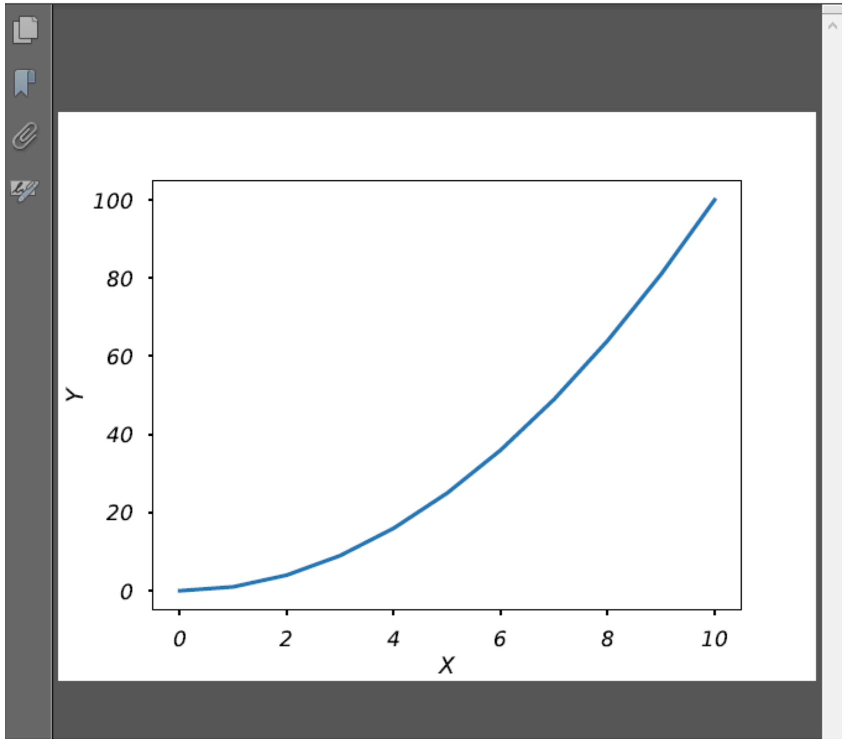

Plot
Scatter
Bar
Loglog
Semilogx
Semilogy
X
Y
0
20
40
60
80
100
0.0
2.5
5.0
7.5
10.0
10^0
10^1
10^2

2D Plotting

- Sometimes, you want to save the figures as a specific format, such as **pdf**, **jpeg**, **png**, and so on. You can do this with the function **plt.savefig**.

In [45]:
```
plt.figure(figsize = (8,6))
plt.plot(x,y)
plt.xlabel('X')
plt.ylabel('Y')
plt.savefig('image.pdf')
```



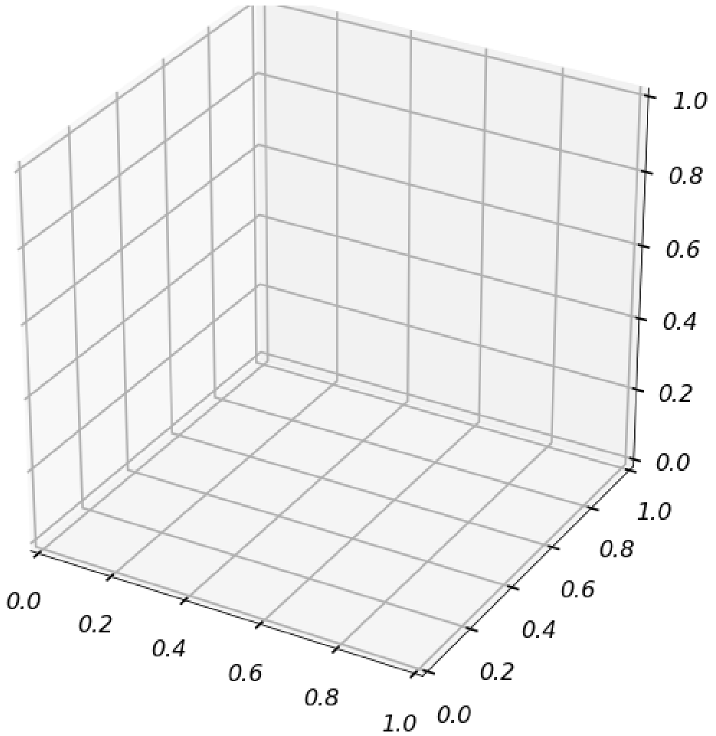


3D Plotting

- In order to plot **3D figures** using **matplotlib**, we need to import the **mplot3d** toolkit, which adds the simple 3D plotting capabilities to matplotlib.

In [46]:
```
import numpy as np
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt
plt.style.use('seaborn-poster')
```

In [47]:
```
fig = plt.figure(figsize = (10,10))
ax = plt.axes(projection='3d')
plt.show()
```

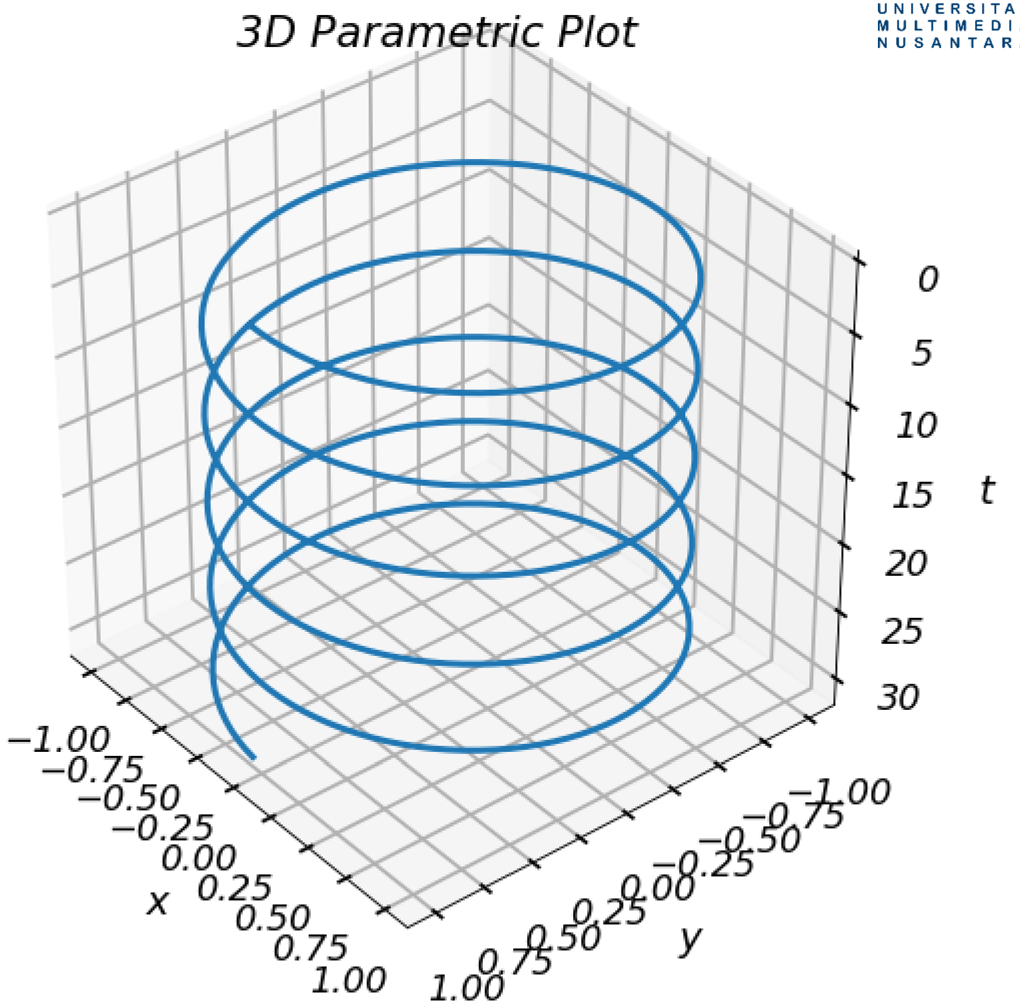


- Once we imported the **mplot3d** toolkit, we could create 3D axes and add data to the axes. Let's first create a 3D axes.

- The **ax = plt.axes(projection='3d')** created a 3D axes object, and to add data to it, we could use **plot3D** function. We could change the title, set the x, y, z labels for the plot as well.

```
In [48]: %matplotlib notebook
```

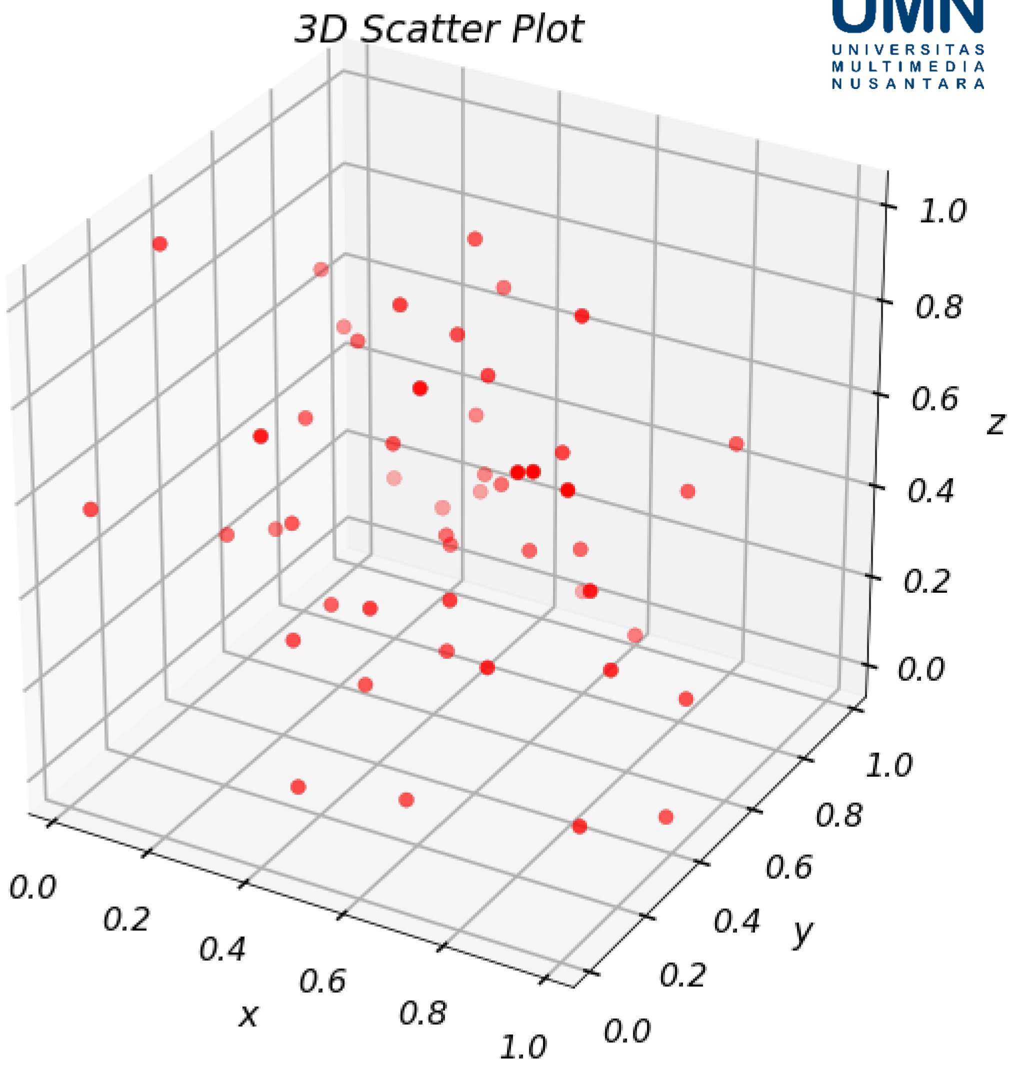
```
In [49]: fig = plt.figure(figsize = (8,8))
         ax = plt.axes(projection='3d')
         ax.grid()
         t = np.arange(0, 10*np.pi, np.pi/50)
         x = np.sin(t)
         y = np.cos(t)

         ax.plot3D(x, y, t)
         ax.set_title('3D Parametric Plot')

         # Set axes label
         ax.set_xlabel('x', labelpad=20)
         ax.set_ylabel('y', labelpad=20)
         ax.set_zlabel('t', labelpad=20)

         plt.show()
```



- Try to rotate the above figure, and get a 3D view of the plot. You may notice that we also set the **labelpad=20** to the 3-axis labels, which will make the label not overlap with the tick texts.

```python
In [50]: # We can turn off the interactive plot using %matplotlib inline
         %matplotlib inline
```

```python
In [51]: x = np.random.random(50)
         y = np.random.random(50)
         z = np.random.random(50)

         fig = plt.figure(figsize = (10,10))
         ax = plt.axes(projection='3d')
         ax.grid()

         ax.scatter(x, y, z, c = 'r', s = 50)
         ax.set_title('3D Scatter Plot')

         # Set axes label
         ax.set_xlabel('x', labelpad=20)
         ax.set_ylabel('y', labelpad=20)
         ax.set_zlabel('z', labelpad=20)

         plt.show()
```



- We could also plot 3D scatter plot using **scatter** function.

Surface Plotting

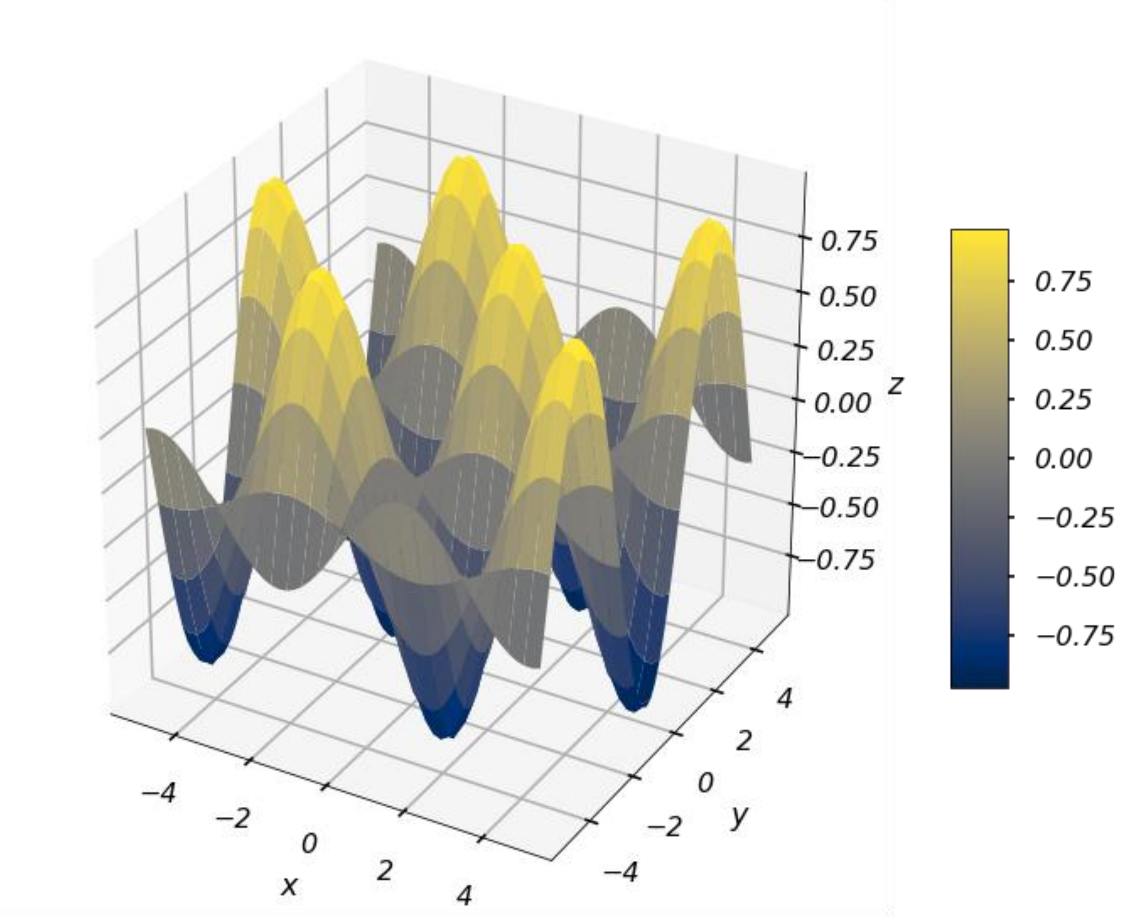
- In **surface** plotting, the first data structure you must create is called a **mesh**. Given lists/arrays of x and y values, a **mesh** is a listing of all the possible combinations of x and y.
- In Python, the mesh is given as two arrays X and Y, where $X(i,j)$ and $Y(i,j)$ define possible (x,y) pairs. A third array, Z, can then be created such that $Z(i,j) = f(X(i,j), Y(i,j))$.
- A mesh can be created using the **np.meshgrid** function in Python. The **meshgrid** function has the inputs x and y are lists containing the independent data set. The output variables X and Y are as described earlier.

In [52]:
```
x = [1, 2, 3, 4]
y = [3, 4, 5]

X, Y = np.meshgrid(x, y)
print(X)
```
```
[[1 2 3 4]
 [1 2 3 4]
 [1 2 3 4]]
```

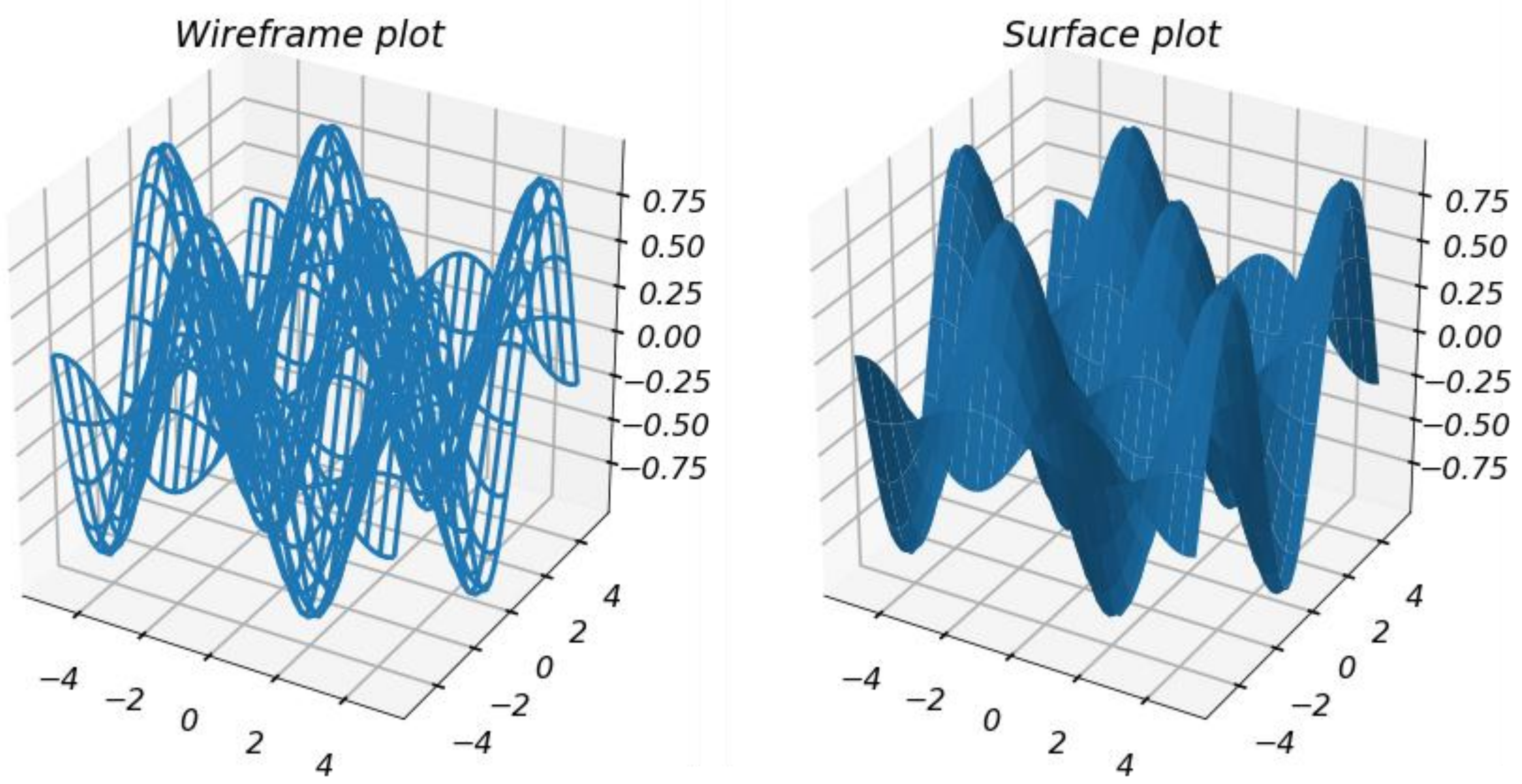
In [53]:
```
print(Y)
```
```
[[3 3 3 3]
 [4 4 4 4]
 [5 5 5 5]]
```

- We could plot 3D surfaces in Python too, the function to plot the 3D surfaces is **plot_surface(X,Y,Z)**, where X and Y are the output arrays from **meshgrid**, and $Z = f(X,Y)$ or $Z(i,j) = f(X(i,j), Y(i,j))$.
- The most common surface plotting functions are **surf** and **contour**.

In [54]:
```
fig = plt.figure(figsize = (12,10))
ax = plt.axes(projection='3d')

x = np.arange(-5, 5.1, 0.2)
y = np.arange(-5, 5.1, 0.2)

X, Y = np.meshgrid(x, y)
Z = np.sin(X)*np.cos(Y)

surf = ax.plot_surface(X, Y, Z, cmap = plt.cm.cividis)

# Set axes label
ax.set_xlabel('x', labelpad=20)
ax.set_ylabel('y', labelpad=20)
ax.set_zlabel('z', labelpad=20)

fig.colorbar(surf, shrink=0.5, aspect=8)

plt.show()
```



```
In [55]:
fig = plt.figure(figsize=(12,6))


ax = fig.add_subplot(1, 2, 1, projection='3d')
ax.plot_wireframe(X,Y,Z)
ax.set_title('Wireframe plot')

ax = fig.add_subplot(1, 2, 2, projection='3d')
ax.plot_surface(X,Y,Z)
ax.set_title('Surface plot')

plt.tight_layout()

plt.show()
```



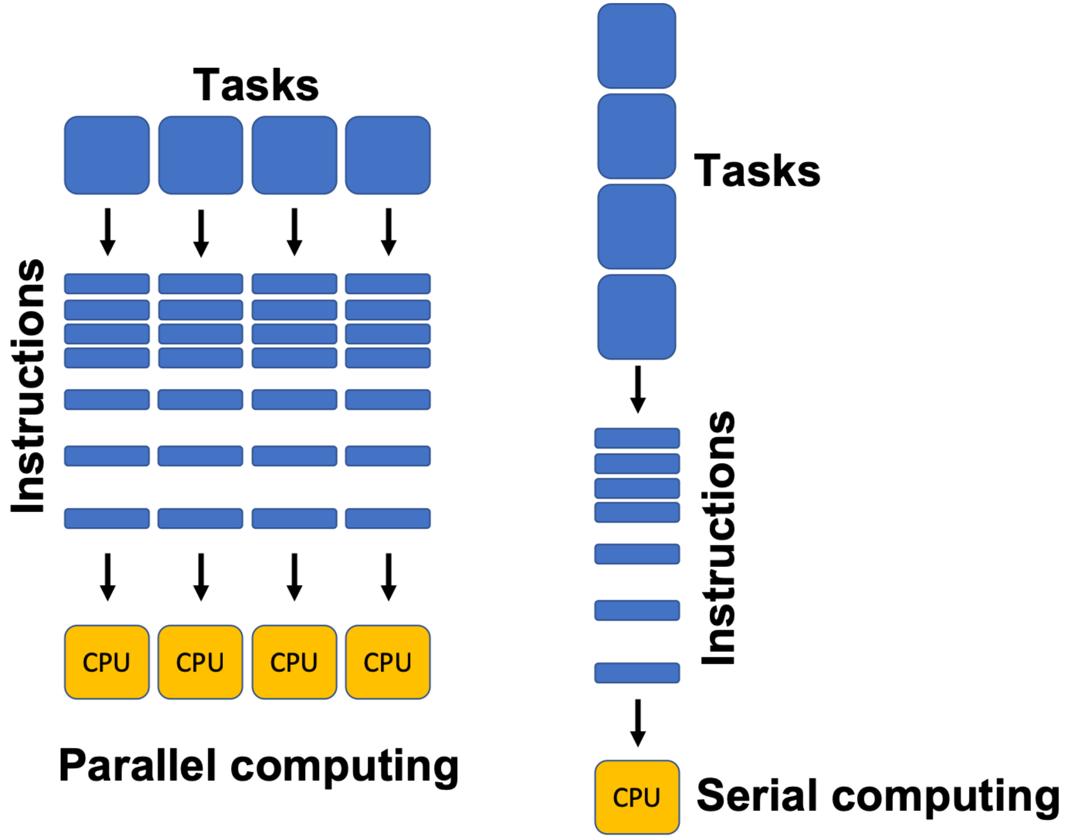
- We could have subplots of different 3D plots as well. We could use the **add_subplot** function from the figure object we created to generate the subplots for 3D cases.

Parallel Your Python

- Now, we will introduce how to run your Python program in **parallel** so that we can reduce the **time** of completing the task.
- For **smaller project**, the benefit may **not** seem **obvious**. But for some **complicated projects**, the **gain** is **significantly**.
- For example, if I give you a homework that normally you need to wait for 1 hour until you can tell whether your code is correct or not. If you find something is wrong, and you change your code accordingly. Now it is time to wait another hour to find out whether your modification is proper. At the end, you made changes for 20 times, and spend more than a day to get it right. But if you can run your algorithms in a parallel way, the running time of your code may shrink to 5 min. 20 times modification only cost you less than 2 hours.
- Now you see why we bother to learn how to run our code faster using the **parallelization**.

Parallel Computing Basics

- The fundamental idea of **parallel computing** is rooted in doing **multiple tasks** at the **same time** to reduce the running time of your program.
- The following figure illustrates the simple idea of doing **parallel computing** versus **serial computing** that we used so far.
- Most of the modern computers are using the **multi-core** design, which means on a single computing component, there are multiple **independent processing units**, the so called **cores**, that are available to do different tasks.



https://pythonnumericalmethods.berkeley.edu/notebooks/chapter13.01-Parallel-Computing-Basics.html

Process and Thread

- In Python, there are **two** basic **approaches** to conduct **parallel computing**, that is using the **multiprocessing** or **threading** library.
- A **process** is an **instance of a program** (such as Python interpreter, Jupyter notebook etc.). A process is created by the operating system to run program, and each process has its own **memory block**. A **thread** is a **sub-process** that reside within the process. Each process can have **multiple threads**, that these threads will share the **same memory block** within the **process**.
- Therefore, for **multiple threads** in a process, due to the shared memory space, the variables or objects are all shared. If you change one variable in one thread, it will change for all the other threads. But things are different in different processes, change one variable in one process will not change the one in other processes.
- **Process** and **thread** both have **advantages** or **disadvantages**, and can be used in different tasks to maximize the benefits.

Python's GIL Problem

- Python has something called **Global Interpreter Lock** (GIL) which allow only **one native thread** to run at a time, it **prevents multiple threads** from running simultaneously.
- This is because Python was designed **before** the **multi-core processor** on the personal computers (this shows you how old the language is).
- Even though there are workarounds in Python to do **multi-threading** programming, we will only cover the **multiprocessing** library in the next section, which we will use most of the time for taking advantage of multi-core parallel computing.
- Of course, there are **disadvantages** of using **parallel computing**. Such as, more **complicated code** and **overheads** when spawn new processes.
- Thus, if your task is **small**, using parallel computing will **take longer** time, since it takes time for the system to **initialize new process** and **maintain** them.

Joblib

- In Python, there are also other 3rd party packages that can make the parallel computing easier, especially for some daily tasks.
- **joblib** is one of them, it provides an easy simple way to do **parallel computing** (it has many other usages as well).
- First you need to install it by running:

```
!pip install joblib
```

- The **Parallel** is a **helper class** that essentially provides a convenient interface for the multiprocessing module. The **delayed** is used to capture the **arguments** of the target function, in this case, the **random_square**.

In [1]:
```
from joblib import Parallel, delayed
import numpy as np

def random_square(seed):
    np.random.seed(seed)
    random_num = np.random.randint(0, 10)
    return random_num**2
```

```
In [2]: import time

t0 = time.time()

results = Parallel(n_jobs=2)\
    (delayed(random_square)(i) for i in range(1000000))

t1 = time.time()
print(f'Execution time {t1 - t0} s')
```

```
Execution time 49.22443435668945 s
```

```
In [3]: t0 = time.time()

results = Parallel(n_jobs=-1, verbose=1)\
    (delayed(random_square)(i) for i in range(1000000))

t1 = time.time()
print(f'Execution time {t1 - t0} s')
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done 24640 tasks      | elapsed:    1.2s
[Parallel(n_jobs=-1)]: Done 205240 tasks      | elapsed:    8.9s
[Parallel(n_jobs=-1)]: Done 506240 tasks      | elapsed:   22.9s
[Parallel(n_jobs=-1)]: Done 927640 tasks      | elapsed:   42.8s
```

```
Execution time 46.2181658744812 s
```

```
[Parallel(n_jobs=-1)]: Done 1000000 out of 1000000 | elapsed:   46.1s finis
hed
```

- We can turn on the **verbose** argument to output the **status** messages.
- There are **multiple backends** in **joblib**, which means using different ways to do the parallel computing.

Next Week's Outline

- Basics of Linear Algebra
- Linear Transformations
- Systems of Linear Equations
- Solutions to Systems of Linear Equations
- Solve Systems of Linear Equations in Python
- Matrix Inversion

References

- Kong, Qingkai; Siauw, Timmy, and Bayen, Alexandre. 2020. Python Programming and Numerical Methods: A Guide for Engineers and Scientists. Academic Press. https://www.elsevier.com/books/python-programming-and-numerical-methods/kong/978-0-12-819549-9
- Other online and offline references

Visi

Menjadi Program Studi Strata Satu Informatika **unggulan** yang menghasilkan lulusan **berwawasan internasional** yang **kompeten** di bidang Ilmu Komputer (*Computer Science*), **berjiwa wirausaha** dan **berbudi pekerti luhur**.

INFORMATIKA
UMN
UNIVERSITAS MULTIMEDIA NUSANTARA

Misi

1. Menyelenggarakan pembelajaran dengan teknologi dan kurikulum terbaik serta didukung tenaga pengajar profesional.
2. Melaksanakan kegiatan penelitian di bidang Informatika untuk memajukan ilmu dan teknologi Informatika.
3. Melaksanakan kegiatan pengabdian kepada masyarakat berbasis ilmu dan teknologi Informatika dalam rangka mengamalkan ilmu dan teknologi Informatika.