Lab 1: Introduction to Jupyter Notebooks and Matplotlib.

DV1604 Interactive Laboratories.

Blekinge Institute of Technology.

Designed and developed by Diego Navaro Tek.Lic, modified by Milena Angelova PhD, 2024.

Description.

The following document compiles the key ideas presented in the first lab of the DV1604 Interactive Laboratories course. It highlights the basic commands and functionality of Jupyter Notebooks, as well as the basics of plotting statistical data using Matplotlib.

The document also offers links to additional external material to help the students expand the available resources that could be useful in the development of the course project.

---

1. Introduction to the Jupyter Notebook.

Jupyter Notebooks is an interactive development platform, originally created to support the *Julia*, *Python*, and *R* programming languages (therefore the acronym *Jupyter*). The idea behind Jupyter notebooks is to offer an effective yet simple way to write, edit, and execute code, simplifying the common tasks involved in data science. Jupyter Notebook is cross-platform, 100% open source, and is widely supported by the academic community and the industry.

> 📚 **Additional Material**: If you wish to know more about the Jupyter project, visit their official Jupyter project website.

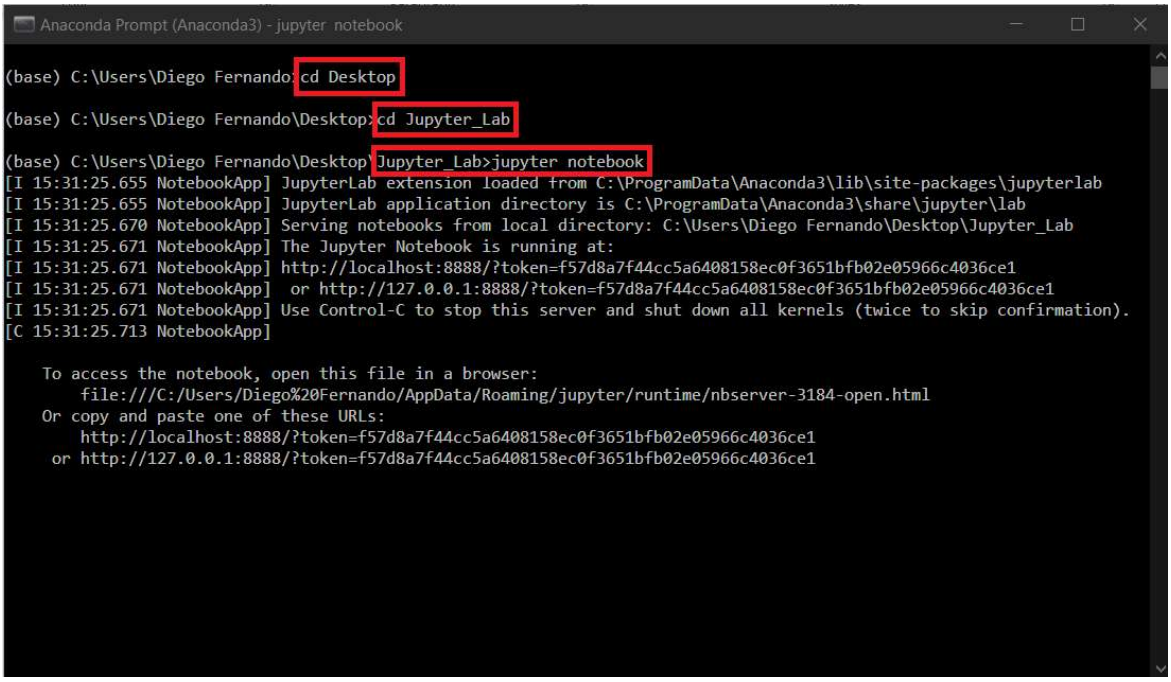
> ⚠️ **Warning:** The following document assumes students are familiar with the Anaconda distribution platform and that they have already installed Jupyter notebooks. If you require help installing Anaconda and/or Jupyter Notebooks, you can find tutorials for it in the *Additional Learning Material* module in *Canvas*.

1.1 Telling Jupyter where to start.

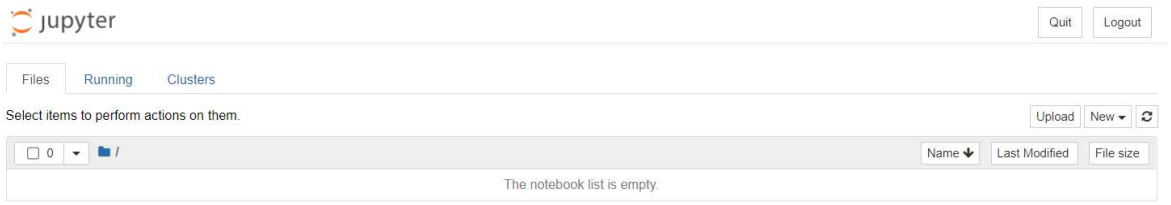
The first thing that we need to do to start using Jupyter Notebooks is to specify where the notebook is going to be located. To do this, open the *Anaconda Prompt* and navigate to the desired location. Remember to use the respective navigation commands for your operative system:

- In Windows, `dir` will display all files and folders in the current directory.
- In Unix systems, `ls` will display all files and folders in the current directory.
- `cd [foldername]` will move you into folder `[foldername]` in your current directory.
- `cd..` will move you out and one level up of the folder you currently are in.

In this example, a folder on the desktop called `Jupyter_Lab` was created. We will tell Anaconda to run Jupyter Notebook inside that folder by using the command `jupyter notebook`:

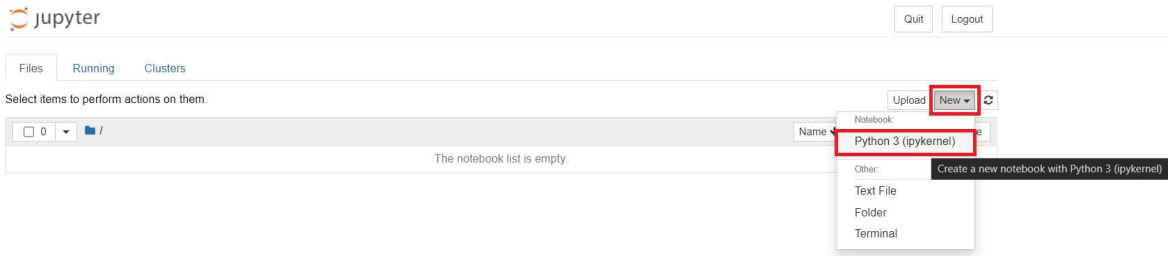


If done correctly a webpage will open using your localhost address, showing the contents of the folder:



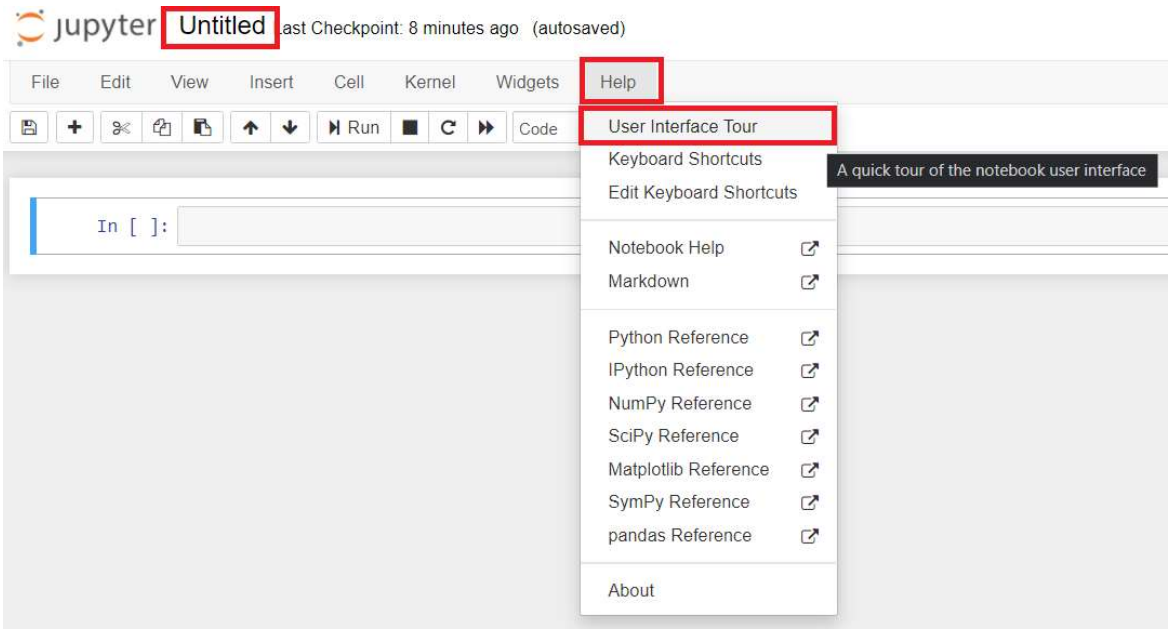
1.2 Creating a Notebook

To create a new notebook for Python programming, click on `New` and then select `Python 3 (ipykernel)` in the top right corner of the file list:



Once you click on `Python 3 (ipykernel)`, the notebook will be created and presented in a new tab on your browser.

The first recommendation is to go to `Help -> User Interface Tour` so you get familiar with how Jupyter Notebook will communicate with you. Pay special attention to the **mode, kernel indicator, and the notification area**. Additionally, you can rename your file by clicking on the `Untitled` name in the top area of your notebook.



Jupyter Notebook features 2 different types of modes: **Edit mode** and **Command Mode.**

- **Edit** is the mode that is enabled by default, every time code is added to a cell in the notebook. Only in Edit mode you can type Python or Markdown code (see Markdown cells in section 1.3). While in edit mode, most of the keyboard shortcuts to control the Jupyter Notebook will be disabled and the selected cell will have a green highlight. To enter Edit mode simply double-click on a cell, or press `Enter`.
- **Command** is the mode that allows you to control the different functionality of the Jupyter Notebook itself. Command is the default mode when no cell is selected. Most of the keyboard shortcuts for Jupyter Notebook only work while in command mode and cells will be highlighted blue when in it. To move from edit to command mode simply press `ESC`. Some of the basic commands while in command mode are:
  - `a` to insert a cell above the current cell.
  - `b` to insert a cell below the current cell.
  - Double-tap `d` to delete a cell.
  - `h` to bring up the Jupyter notebook cheat sheet.

1.3 Using the Jupyter notebook.

To use the Jupyter Notebook, 2 different types of cells are provided to the user: **Code** and **Markdown** cells.

Code Cells.

Code cells are the default type of cell generated by the Jupyter Notebook. In this cell, Python code can be directly typed and executed, just like in any other python IDE (e.g. you can do direct computations, create variables, loop operations, import libraries, create functions, etc.).

In [ ]:
```
2 + 3 + 4
```

In [ ]:
```
a = 20
b = a +50
print(b)
```

In [ ]:
```
for i in range(5):
    print(i)
```

Two different keyboard commands are available in Jupyter Notebook to run the code in a cell:

- `ALT + Enter` will execute the code in the selected cell, and it will add a new cell below.
- `CTRL + Enter` will execute the code in the selected cell, but no new cell will be inserted below.

The `In[ ]`: prefix that appears before the cell frame shows that Jupyter notebook understands the code as an input. Similarly, `Out[ ]`: shows the output generated by the Jupyter Notebook's kernel. Lastly, the number that appears

the between the squared brackets shows the order in which each cell has been executed.

> 💡 **Tip**: Since the ipython shell allows for *concurrency* (to run code out of order), the Jupyter Notebook keeps track of the code execution order.

Markdown Cells.

Markdown cells allow you to insert plain text into your notebook. It is based on the *Markdown* markup language, which is a syntax for plain text formatting. It is commonly used in readme files, websites (such as Github, Twitch, or Discord), online forums, and some forms of XML.

To turn a Code cell into a Markdown cell, you need to select the cell and press `m` on command mode (remember, to exit edit mode after putting data into a cell by pressing `ESC`). Then press `CTRL + Enter` or `ALT + Enter` to execute the markdown syntax.

Here are some basic examples of how to use Markup syntax:

1. Headings: The size of a heading can be changed using the `#` character. The more `#` characters, the smaller the heading.

```
# Headings
```

```
# Biggest Heading
```

Biggest Heading

```
###### Smallest Heading
```

Smallest Heading

```
Normal text
```

2. Text emphasis: By using the `*` character, the emphasis of text can be changed between *italic*, **bold**, or ***bold and italic***.

```
# Emphasis
```

```
## Bold: This is a **bold text** example
```

Bold: This is a **bold text** example.

```
## Italic: This is a *italic text* example
```

Italic: This is an *italic text* example.

```
## Bold and Italic: This is a ***bold and italic*** example
```

Bold and Italic: This is a ***bold and italic*** example.

3. Lists: Lists are automatically formatted when using a `Number`, or the `*`, `-`, and `+` characters to list elements.

```
# Lists
```

```
## Numbered List
```

```
1. One
2. Two
3. Three
4. Four
```

1. One
2. Two
3. Three
4. Four

```
## Bullet list
```

```
- One
- Two
- Three
- Four
```

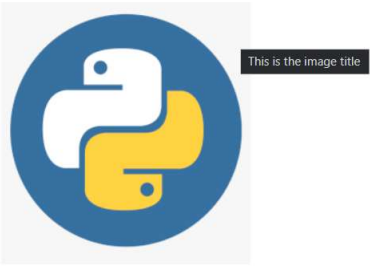
- One
- Two
- Three
- Four

4. Images: The `!` character defines the following inputs as an image. You can add alternative text for the image between squared brackets `[ ]`, and the relative location of the image between parenthesis `( )`. Additionally,

image titles can also be added in the parenthesis by using quotation marks `" "`.

Images

```
![This is alternative text](python.png "This is the image title")
```



1.4 Additional resources about the Markdown language.

Markdown is a very versatile language for formatting text. Since this lab only covers its basics, **make sure to review the multiple options it features**. You can review:

- A basic Markdown cheat sheet.
- The Markdown basic syntax.
- The Markdown extended syntax.
- Markdown Ninja Hacks.

> 💡 **Tip**: Markdown supports the use of some `HTML` commands. If a particular feature is not natively supported by Markdown (e.g. changing text color, changing image size, etc.), you can check if there is an `HTML` command that can complete the task.

---

2. Data Visualization in the Jupyter Notebook using Matplotlib.

Matplotlib is one of the most popular and robust libraries for plotting statistical data using Python. It can implement a wide variety of data representation techniques, 3D plotting, or even animated plots. Matplotlib's functionality is based upon 2 main concepts: *Figures* and *Axes*:

- The **Figures** can be understood as containers, in which the code for visualizing data goes. Figures can have properties and variables that can directly affect all the elements kept inside the figure. Think of it as the canvas in a painting.
- The **Axes**, also referred to as *Subplots*, refer to the actual code that generates the plot, as well as the variables that define the plotting limits. Think of it as the paint in a painting.

In Matplotlib, all Axis/Axes (that are treated as objects) are contained within a Figure. In this manner, it is easier to apply different methods to single or multiple Axes within the figure (similar to how an entity-component system work).

> 💡 **Tip**: There are two different interfaces to control Matplotlib: *Object-Oriented* and *Pyplot*. The Pyplot interface is a collection of functions that aim to provide a MATLAB-like syntax to use Matplotlib, simplifying the way some commands are written. For the development of this course, we will use the Pyplot interface.

> ⚠️ **Warning**: Anaconda usually makes sure that all the packages installed in your system are up to date. However, there are cases in which Anaconda fails to install or update Matplotlib. If you experience any issues running any of the following excercises, make sure that you have Matplotlib properly installed in your system by executing the `conda install -c conda-forge matplotlib` command in the Anaconda Promt, followed by the `conda update --all` command.

2.1 Setting-up Figures and Axes in Matplolib.

First of all, Matplotlib needs to be prepared for rendering the visualizations we implement inside the Jupyter Notebook. For this, the magic command `%matplotlib inline` is used. This command prevents Matplotlib from

creating a new window/tab to display the figures we program and, instead, uses a Notebook cell to directly render our plots.

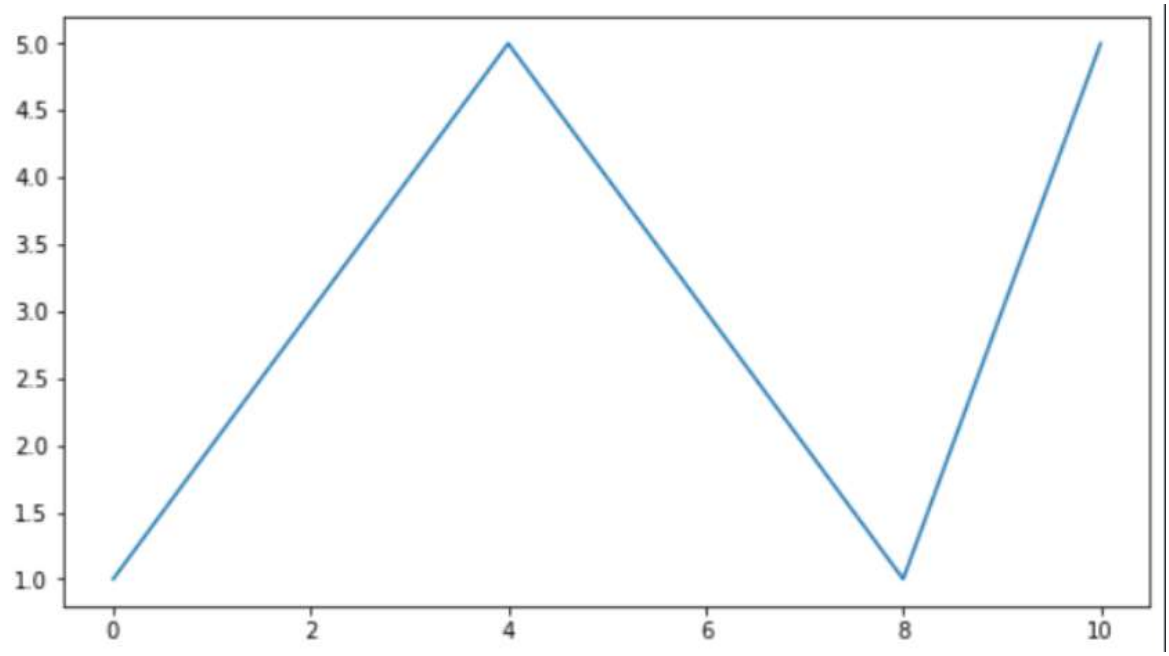
> ⚠ **Warning**: Because of this, **the `%matplotlib inline` command must be written before Matplotlib is imported in the Notebook**.

> 💡 **Tip**: *Magic commands* (also known as *magics*) are built-in commands in the IPython kernel that allow modifying the native properties or behavior of the kernel. It is a common method to fastly change how the kernel performs certain operations. For more information, you can review the IPython Built-in Magic Commands documentation.

To import Matplotlib we use the command import `matplotlib.pyplot as mpl`.

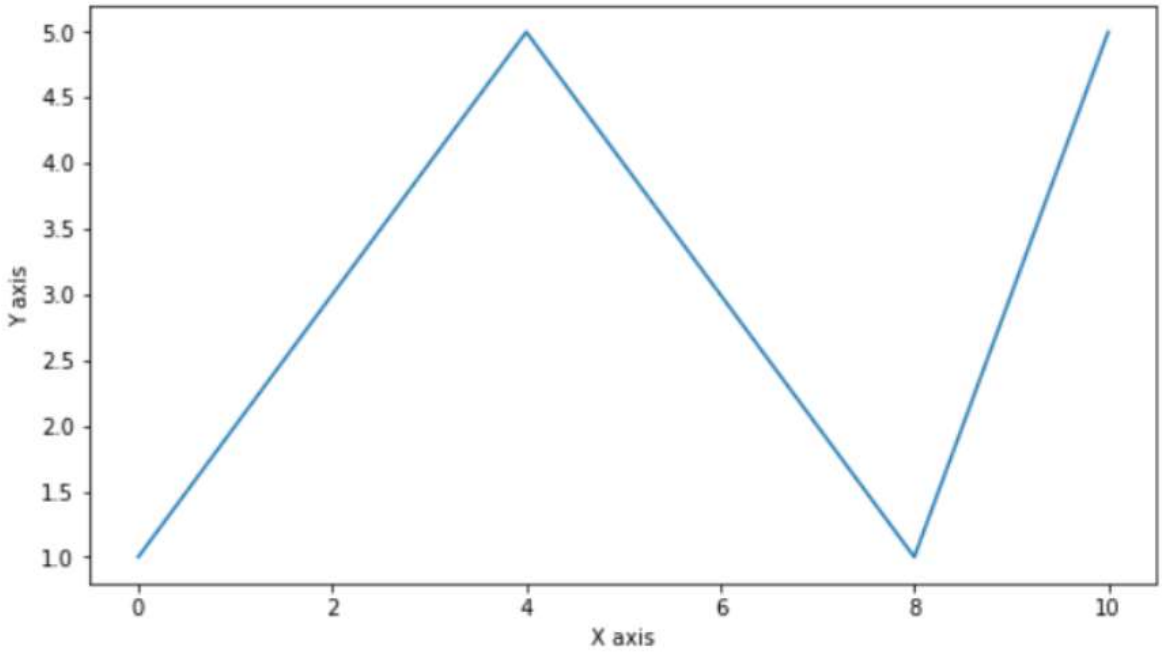
In [ ]:
```
#Magic command
%matplotlib inline

#Matplotlib import
import matplotlib.pyplot as mpl
```

Once the library is loaded, we can start plotting data with Matplotlib. For this, we need to first create a Figure using the `mpl.figure` command, and the Axes using the `add_subplot` command. In the following example, the Figure is stored within the `fig` variable, and the Axes within the `ax` variable. Lastly, we can plot data using the `ax.plot` command. By default, a **Line plot** is used by `ax.plot`:



In [ ]:
```
#createing ficticious data to plot.
data = [1,2,3,4,5,4,3,2,1,3,5]

# add the code here
```

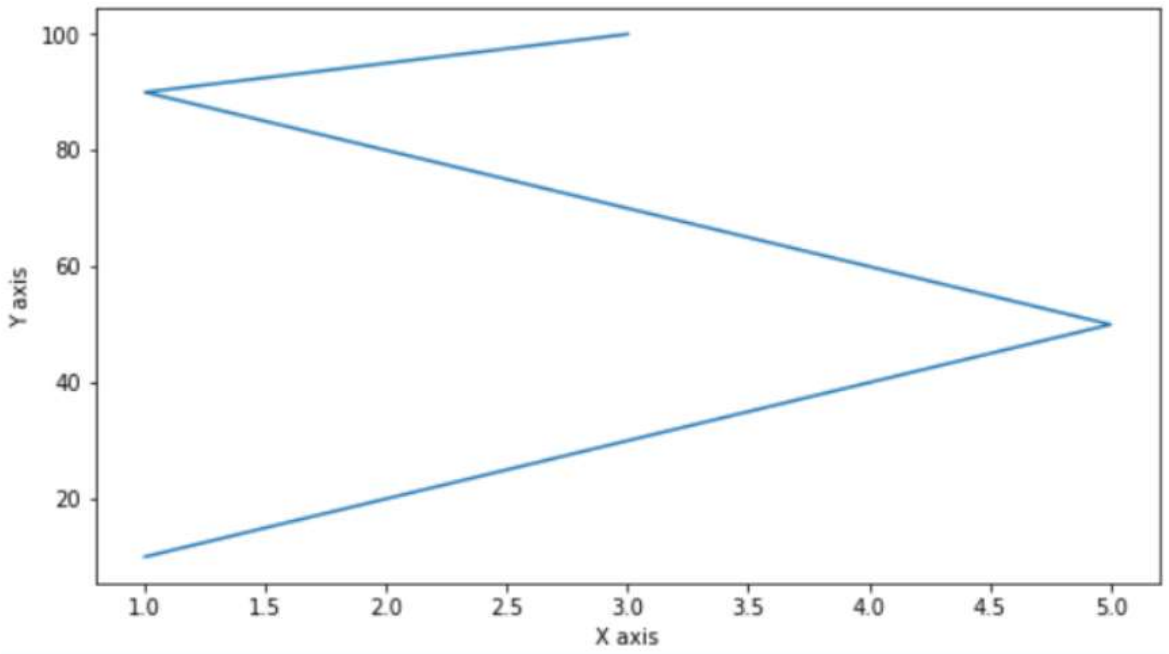
To add labels to the plot axes, we add the `ax.set_xlabel` and `ax.set_ylabel` commands, specifying the name of each respective axis in between quotation marks `" "`:



In [ ]:
```
data1 = [1,2,3,4,5,4,3,2,1,3]
data2 = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# add some code here
```

The previous examples use the information stored in `data` as a coordinate list for the Y axis. However, the `ax.plot` command can support the use of many arguments. For example, we can provide an additional list in the plot command, using the data in `data1` for the X axis, and the data in `data2` for the Y axis.

> **Additional Material**: for more information review the matplotlib.pyplot.plot documentation.

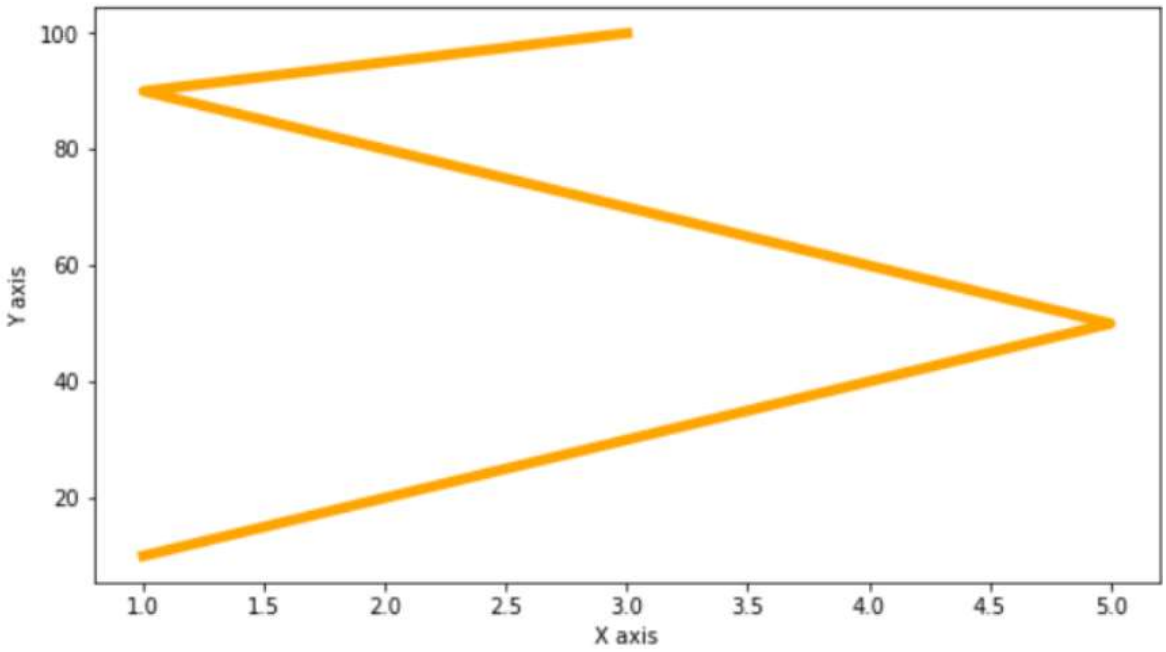


In [ ]:
```
data1 = [1,2,3,4,5,4,3,2,1,3]
data2 = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# add some code here
```

As you can start inferring, creating plots with Matplotlib involve an iterative process of adding parameters to the plot command and the plot Axes. It is because of this that Jupyter Notebooks comes in handy. You can directly modify each of the properties of your plots and execute only the cells where updates were made. Now, let us modify some of the attributes of the current plot we have.

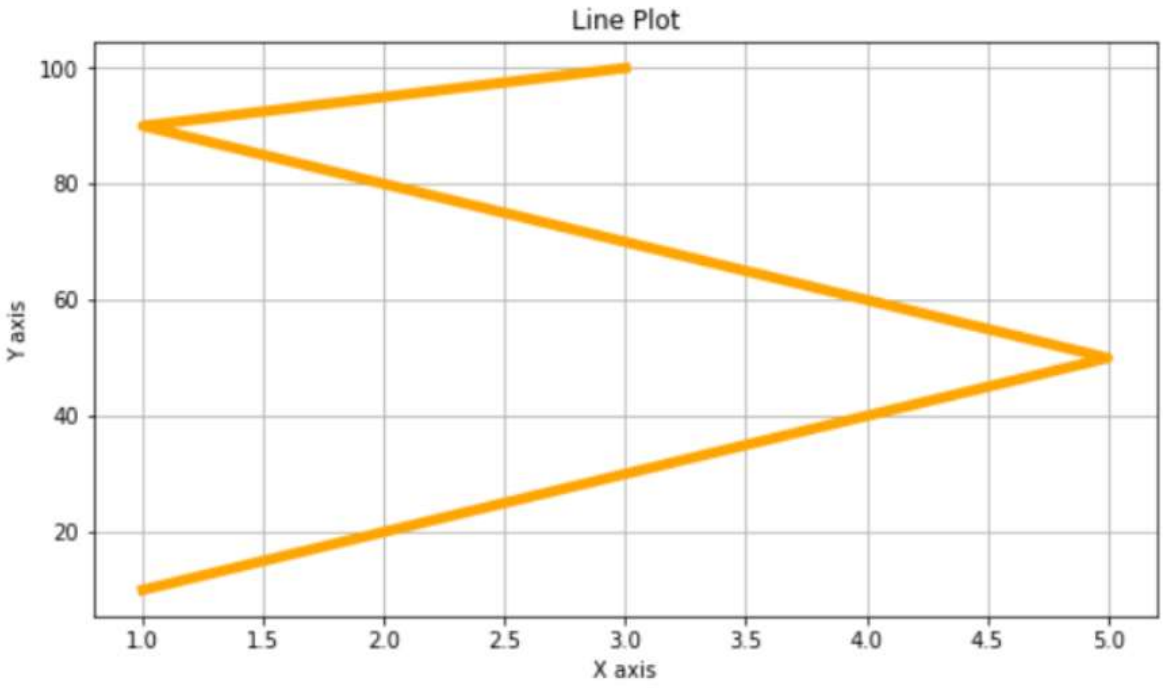
The `color` and `linewidth` attributes of the plot can be changed by adding data to those parameters in the plot command. For `color` we simply write the color we want between quotation marks, while `linewidth` will expect a scaling number.

> **Additional Material**: For a full reference of the supported colors in Matplotlib, check the List of Named Colors documentation.



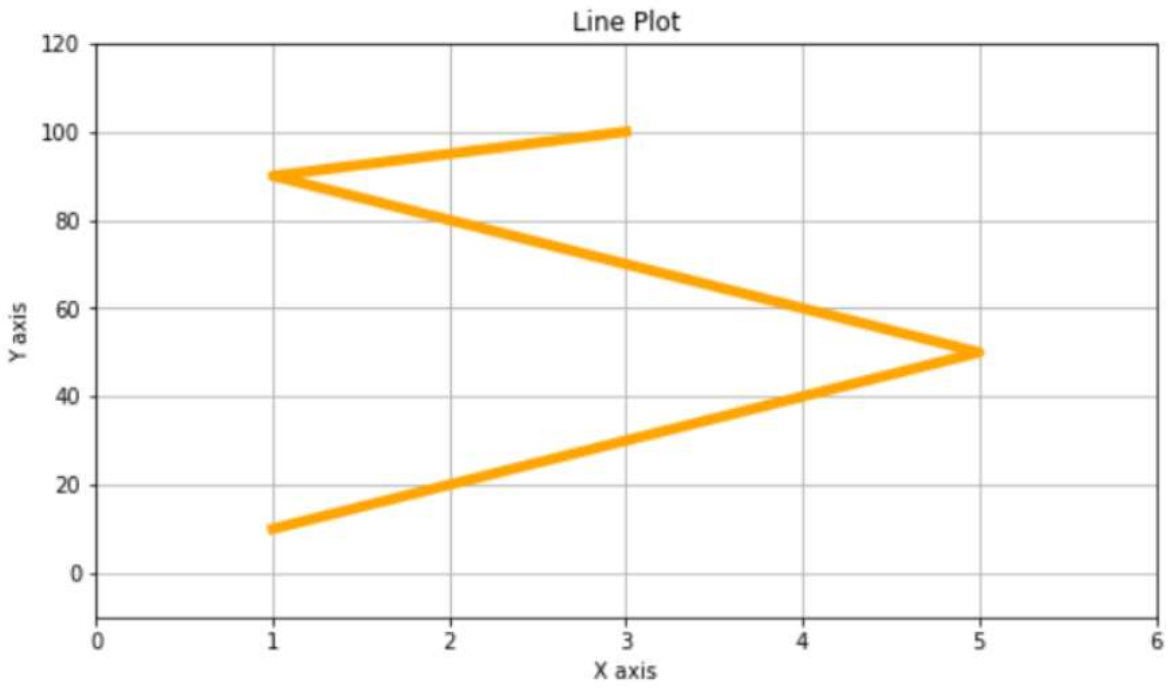
In [ ]:
```
data1 = [1,2,3,4,5,4,3,2,1,3]
data2 = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# add the code here
```

We can assign a plot title with the `ax.set_title` command, specifying the name (*Line Plot* in this example) between quotation marks. Also, we can establish a grid for our plot to improve the readability of the plot using the `ax.grid` command.



In [ ]:
```
data1 = [1,2,3,4,5,4,3,2,1,3]
data2 = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# add the code here
```

To modify the range of data covered by each axis, we can establish limits for the X and Y axes with `ax.set_xlim` and `ax.set_ylim` commands. Even if Matplotlib will automatically set the data ranges to fit the render area, we can customize the way the data is being presented by altering the limits of the axes:



In [ ]:
```
data1 = [1,2,3,4,5,4,3,2,1,3]
data2 = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# add the code here
```

2.2 Different data representations with Matplolib.

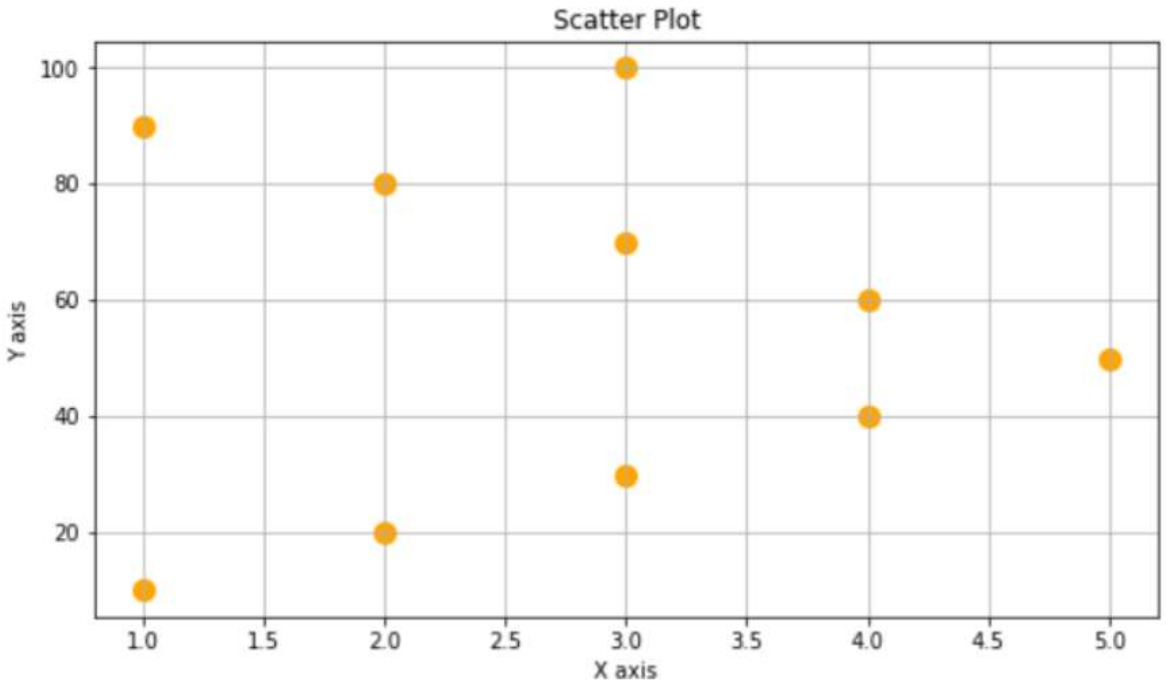
Matplotlib offers multiple ways of representing data. In this opportunity, we will focus on implementing the data representations that were considered key in data analysis during the *Principles of Data Visualization* lecture: line plots (which we already covered above), scatter plots, bar/histograms plots, and box plots. In addition to this, the basics of plotting mathematical functions, multiplots, and 3D plotting will also be covered in this lab.

It is important to remember that each data representation technique is useful in one specific scenario (comparing categorical variables, showing trends over time, highlighting data distributions, etc.), so make sure you present the right data, in the right manner, when choosing a data representation.

Scatter Plot

To implement a scatter plot the command `ax.scatter` is used. Just like in the previous example we can modify attributes like the color and the size of the points with the linewidth command.

> **Additional Material**: For more information review the matplotlib.pyplot.scatter documentation.

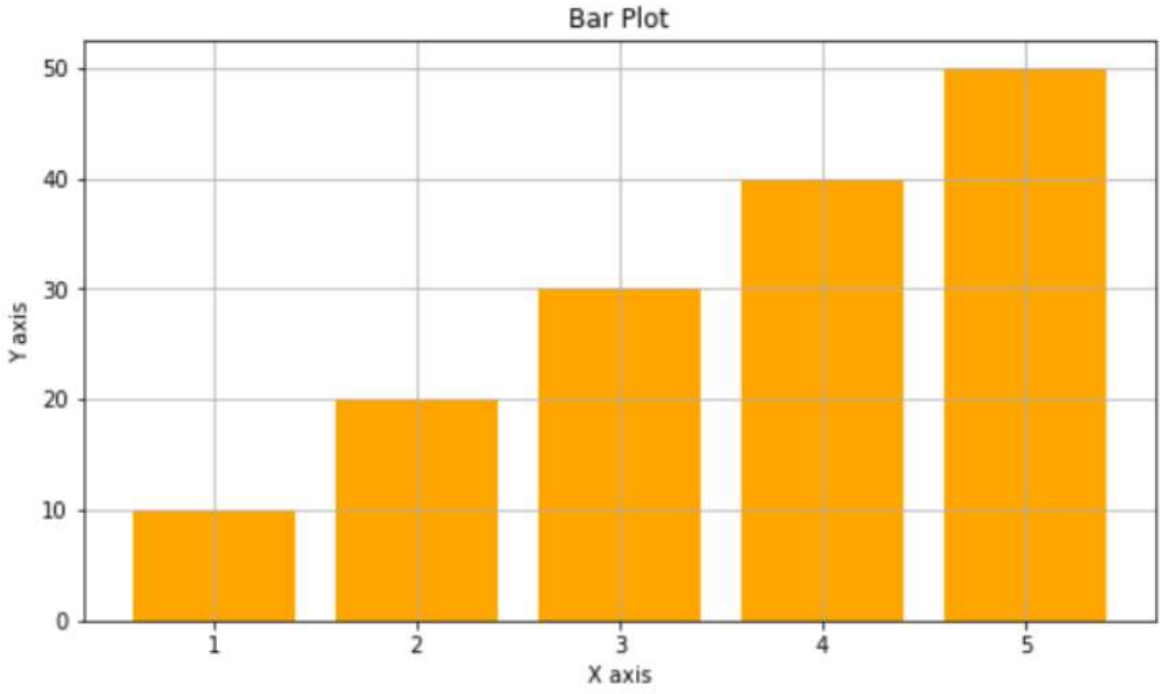


In [ ]:
```
data1 = [1,2,3,4,5,4,3,2,1,3]
data2 = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]

# add the code here
```

Bar Plot

The bar plot is implemented using the `ax.bar` command. For the following example, `data1` represents the data for the X axis, while `data2` is for the Y axis.

> **Additional Material**: For more information review the matplotlib.pyplot.bar documentation.



In [ ]:
```
data1 = [1,2,3,4,5]
data2 = [10, 20, 30, 40, 50]

# add the code here
```

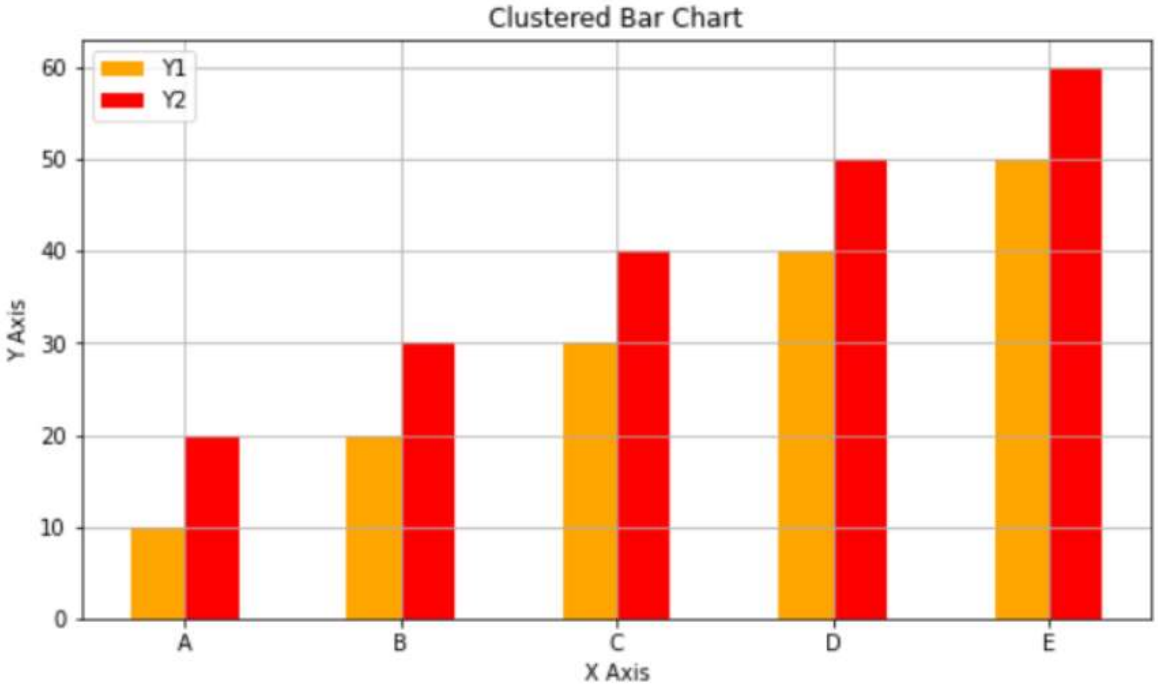
Clustered Bar Plot

To implement a clustered bar plot, we must modify some of the parameters in the bar plot, since Matplotlib does not offer a dedicated method for generating this kind of data representation. For this example, we are going to use `data_x` for the X axis, and two datasets for the Y axis, `data_y1` and `data_y2` respectively.

In clustered bar charts, Matplotlib will not automatically rearrange the data used in the Y Axis with the data in the X Axis (if we do this directly we may have overlapping issues). We need to manually indicate to Matplotlib how the data needs to be rearranged in the plot. First, we will create a list and store the labels for the X Axis in `x_labels`. Then, we will use the `np.arange` method from *Numpy* to retrieve an evenly spaced range of values based on the amount of labels in the X axis, store it in `data_x`, and use it to reference a position for the data stored in `data_y1` and `data_y2`.

Finally, we set a width for our bars in `width`. In each `ax.bar` command, we move each bar `width/2` to the left and the right respectively, to solve the overlapping issue. Last, we modify the labels that are displayed in the X axis using the `ax.set_xticks` command and data stored in `x_labels`.

> 💡 **Tip**: *Numpy* is a Python library that provides a multidimensional array object and an assortment of routines for fast operations on arrays, including mathematical, logical, sorting, selecting, basic linear algebra, basic statistical operations, among other applications. For more information visit the Numpy official documentation.



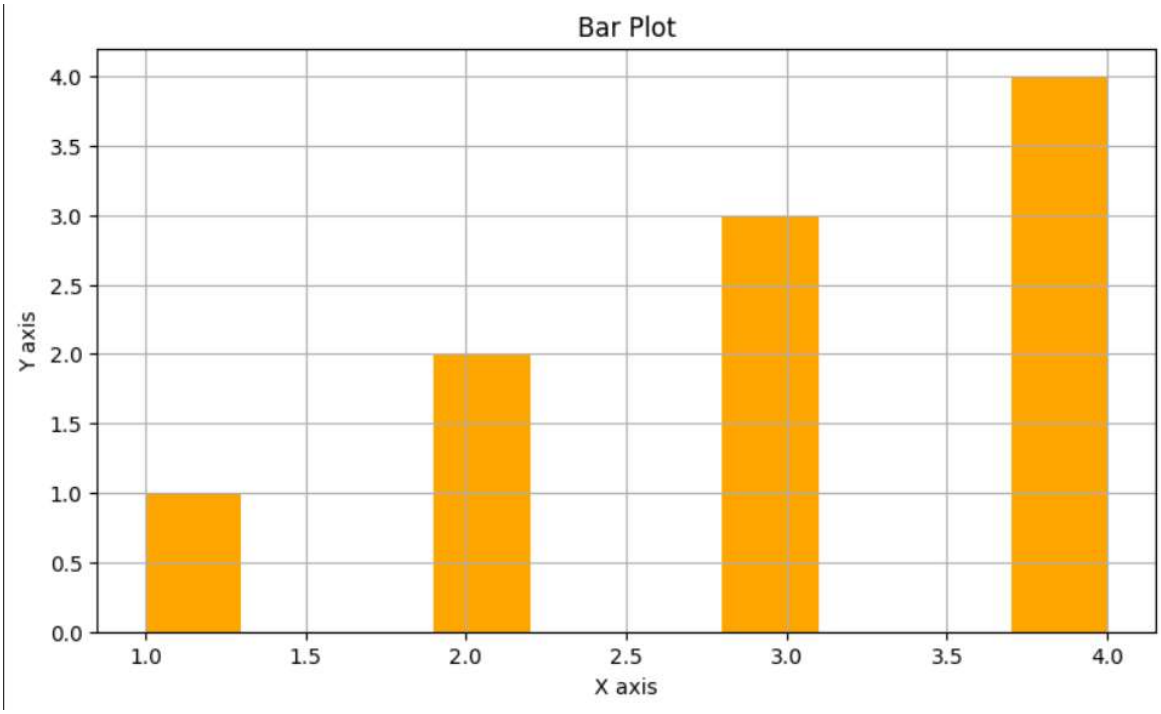
In [ ]:
```
#importing Numpy
import numpy as np
```

In [ ]:
```
x_labels = ["A","B","C","D","E"]
data_y1 = [10, 20, 30, 40, 50]
data_y2 = [20, 30, 40, 50, 60]

#Creating a range for the X axis
data_x = np.arange(len(x_labels))

# add the code here
```

Histogram

The histogram plot represents the *frequency* (the amount of times the same value for a data point repeats itself) in the data. Therefore, it only requires one single array of data. The `ax.hist` command will automatically calculate the frequency of the data and plot the respective values.

> 📚 **Additional Material**: For more information review the matplotlib.pyplot.hist documentation.

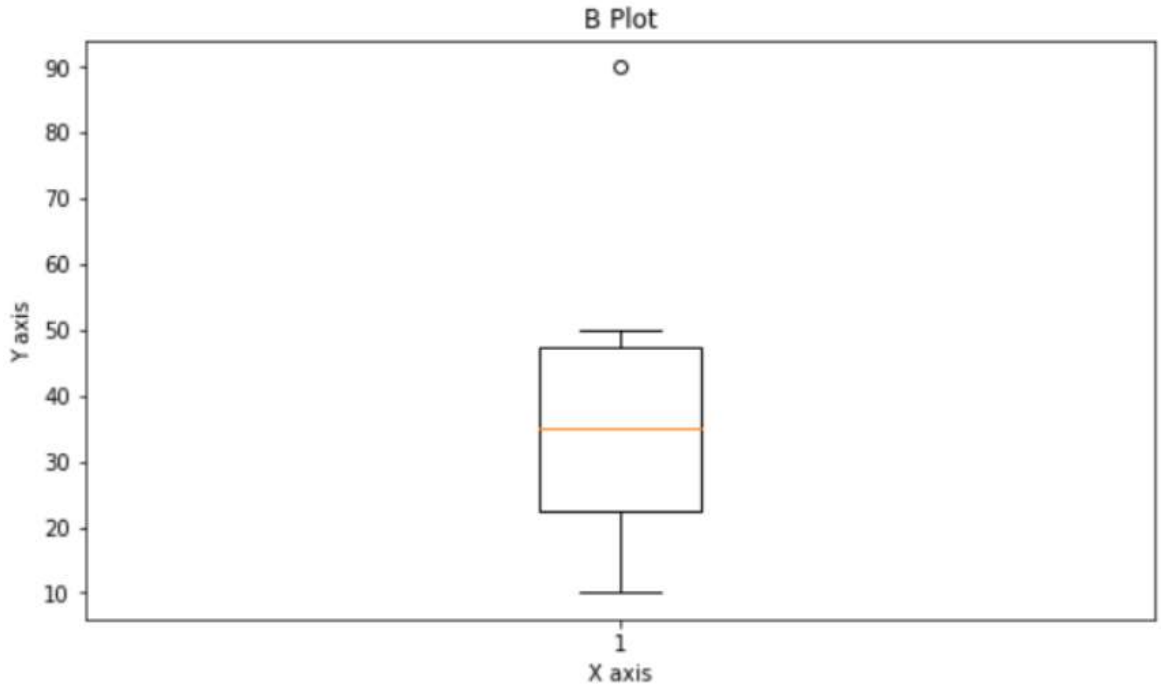


In [ ]:
```
data1 = [1,2,2,3,3,3,4,4,4,4]

# add the code here
```

Box Plots

Box plots are a representation of data distribution. Therefore, it requires only one array of data to generate the plot. In the example, an *outlier* (a data point far off from the rest of the distribution) was intentionally placed to highlight the way they look. Also, the *median* (data point that is in the middle of the distribution) is shown in orange color in the box plot. To generate a box plot we use the `ax.boxplot` command.

> **Additional Material**: For more information review the matplotlib.pyplot.boxplot documentation.



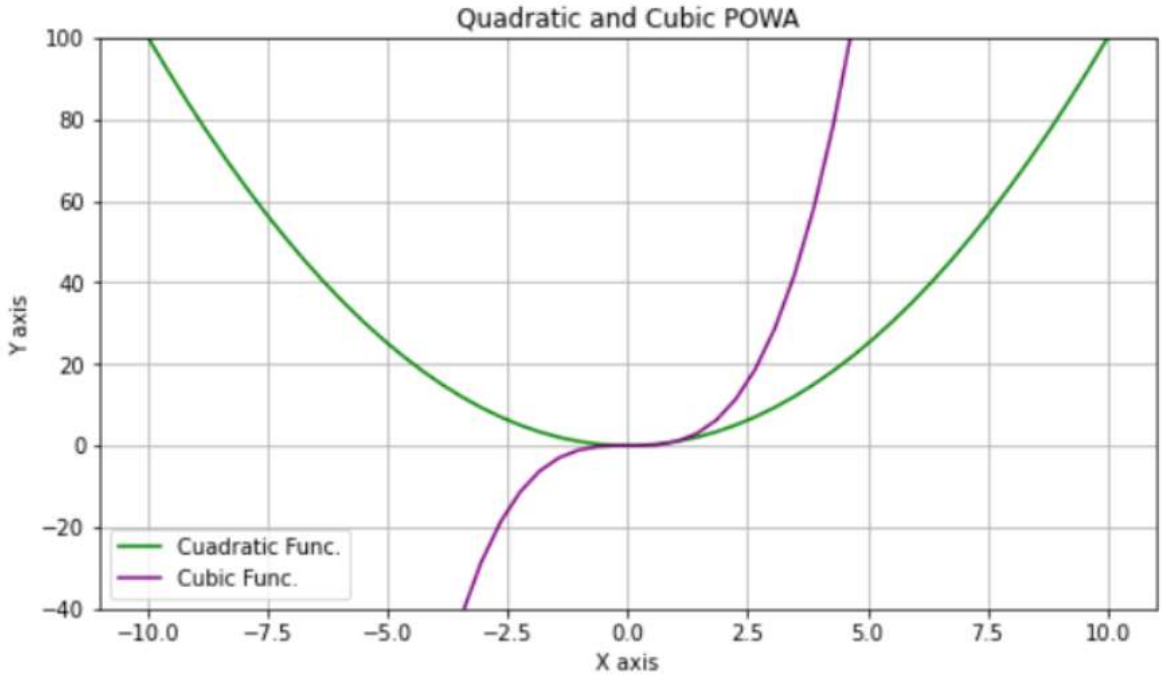
In [ ]:
```
data1 = [10, 20, 30, 40, 50, 90]

# add the code here
```

Mathematical Functions in Matplolib

The capabilitites of Matplotlib are not limited to the different data representation functions that have been reviewed so far. The library can also be used to plot raw mathematical functions. For the following example we will plot the functions $y = x^2$ and $y = x^3$.

First, a range of values between -10 and 10 is stored within `xrange`, using the Numpy library and the function `np.linspace`, for the X axis. Then, we create 2 variables for the Y data, one for a quadratic power `y1`, and another for a cubic power `y2`.

Lastly, since we need to differentiate `y1` from `y2` within our plot, we need to implement a *legend*. Legends are small inserts that clarify the type of data, or the variable names, that are being shown in the plot. To enable a legend we simply use the command `ax.legend()`, and to customize the text shown by the legend, we must set up the `label` parameter in each `ax.plot` command.

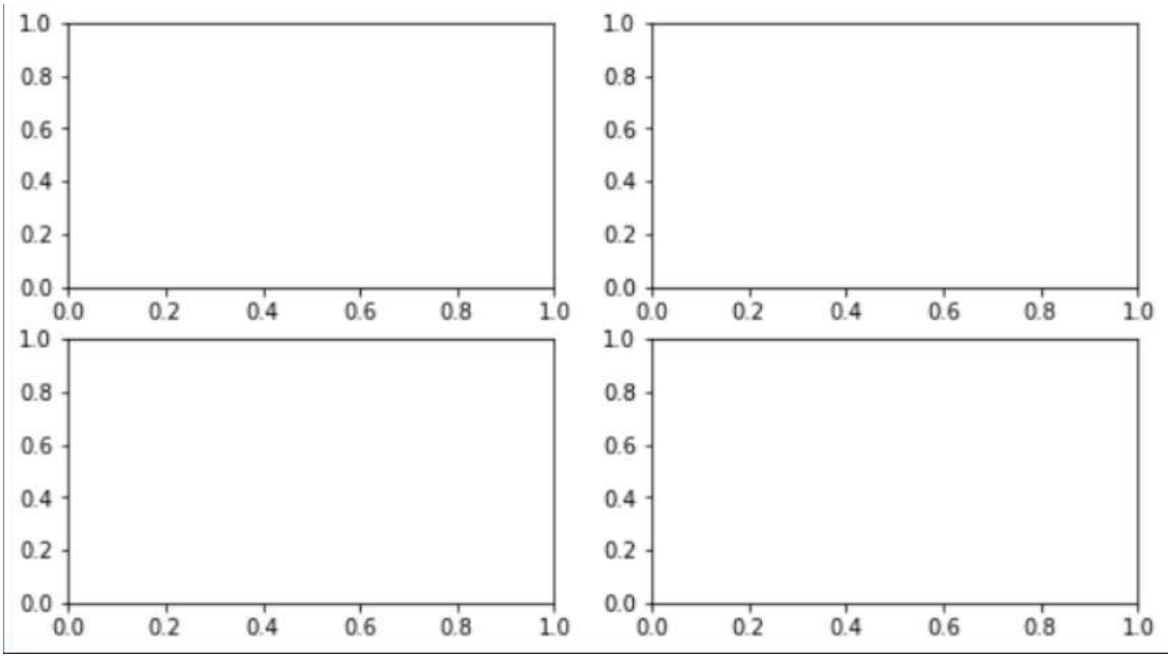


In [ ]:
```
# data range for the X axis
xrange = np.linspace(-10.0, 10.0)

#data for the y axis
y1 = xrange**2
y2 = xrange**3
```

In [ ]:
```
fig = mpl.figure(figsize=(9,5))

# add the code here
```

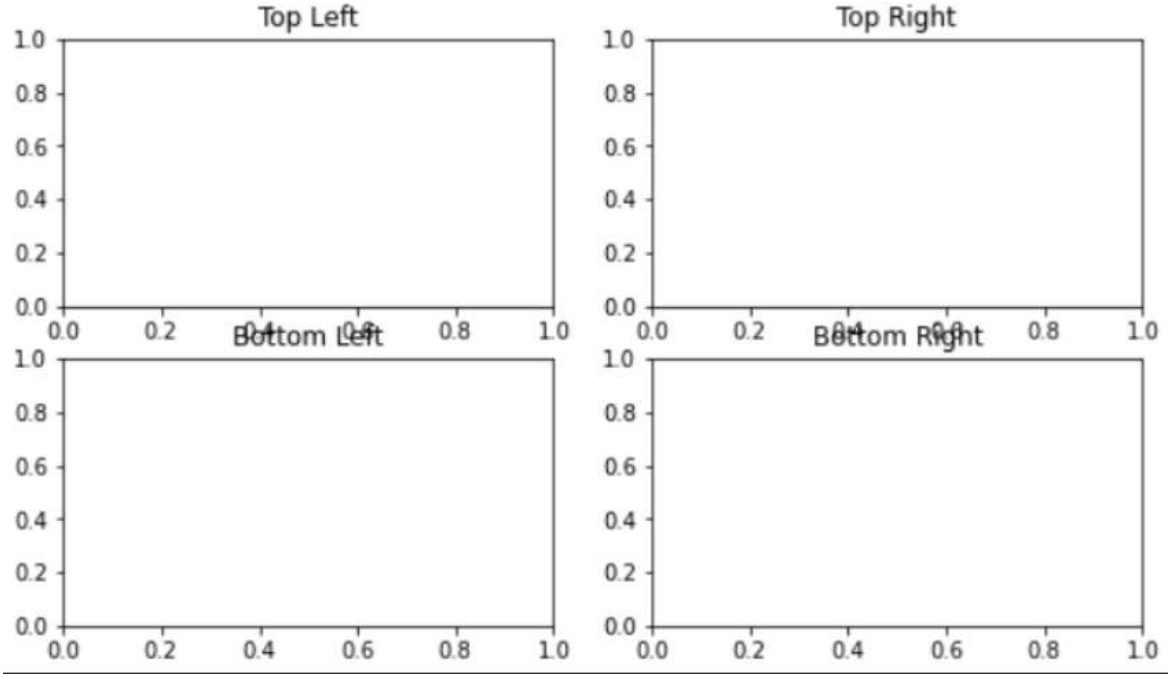
Multiplots

Multiplots are Figures that render multiple Axes simultaneously. It is commonly used when multiple data sets, or different types of data, need to be analyzed together. To implement a Multiplot we require a `fig` container and the Axes. The difference is that we are going to use a matrix, `axes`, to manipulate our Axes. We use the command `fig.subplots(nrows=2, ncols=2)` to turn `axes` into a 2x2 matrix object that controls 4 different subplots:



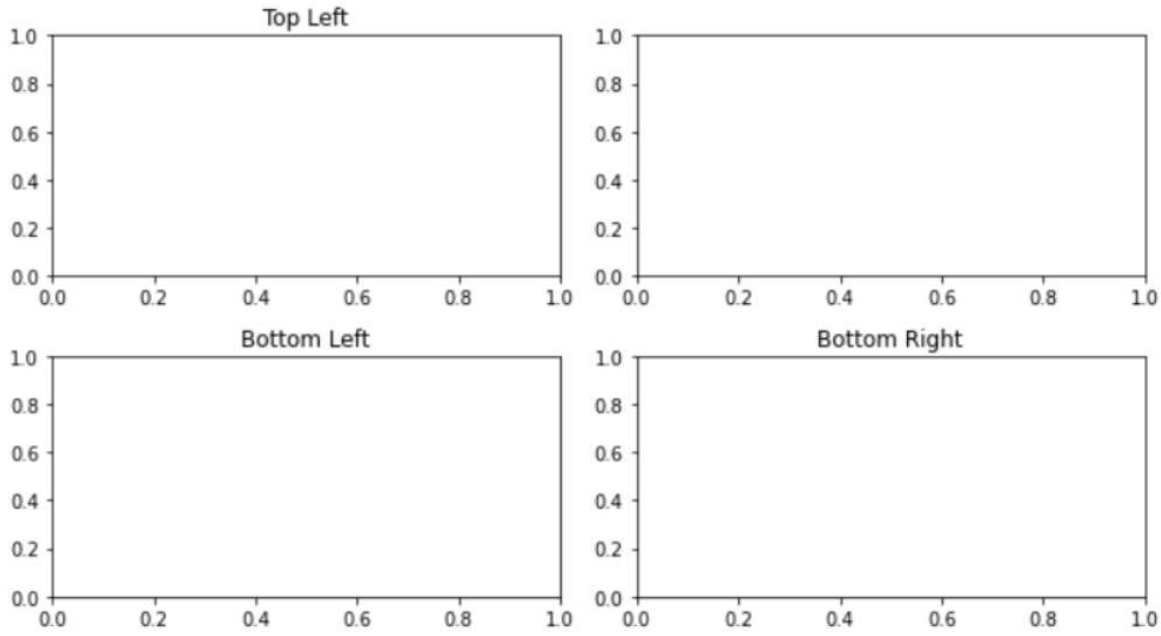
In [ ]:
```
fig = mpl.figure(figsize=(9,5))
# add the code here
```

To understand how the indexing of `axes` controls the plots, we will assign an explicit name for each plot:



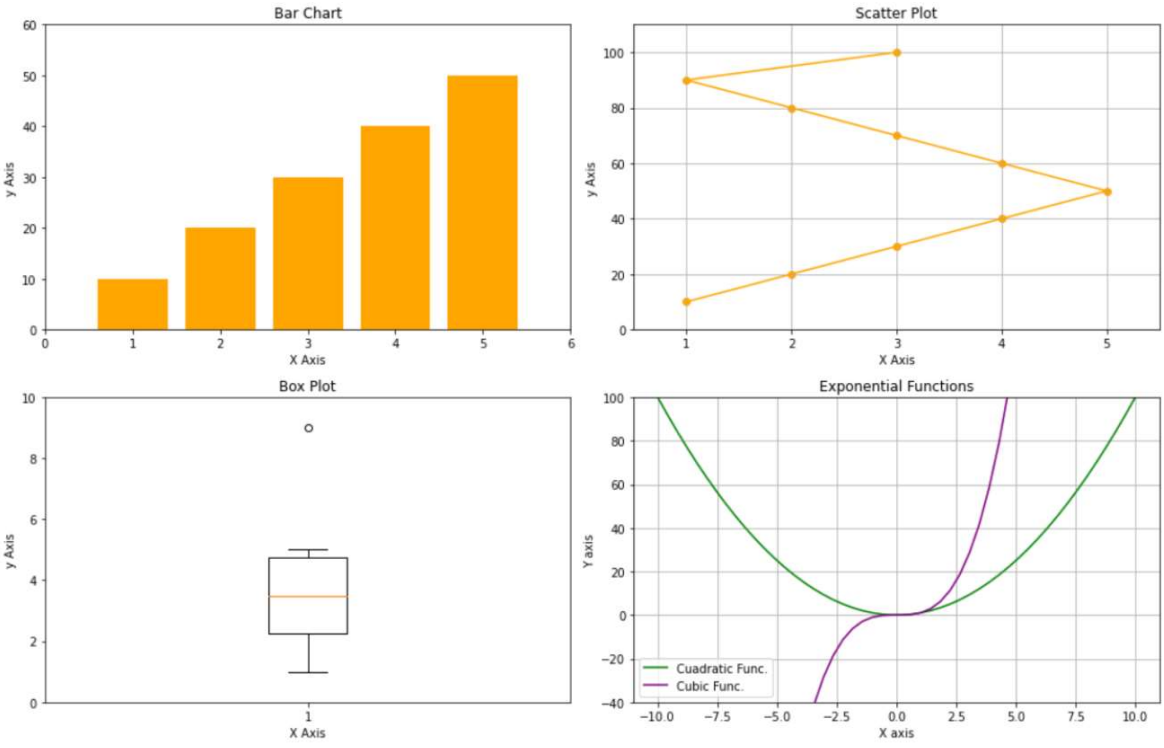
In [ ]:
```
fig = mpl.figure(figsize=(9,5))
# add the code here
```

A problem that we usually run into is the overlapping of axes and labels. To address this issue, we need to tell `fig` to adjust the position of each Axis, so axes and labels won't overlap. For that we use the function `fig.tight_layout()`:



In [ ]:
```
fig = mpl.figure(figsize=(9,5))

# add the code here
```

Now, with our Axes ready, we can plot any function we need. For this example we will reuse some of the previous code and plot 4 different data representation techniques, adding labels, titles, data ranges, and modifying the attributes of the plots:



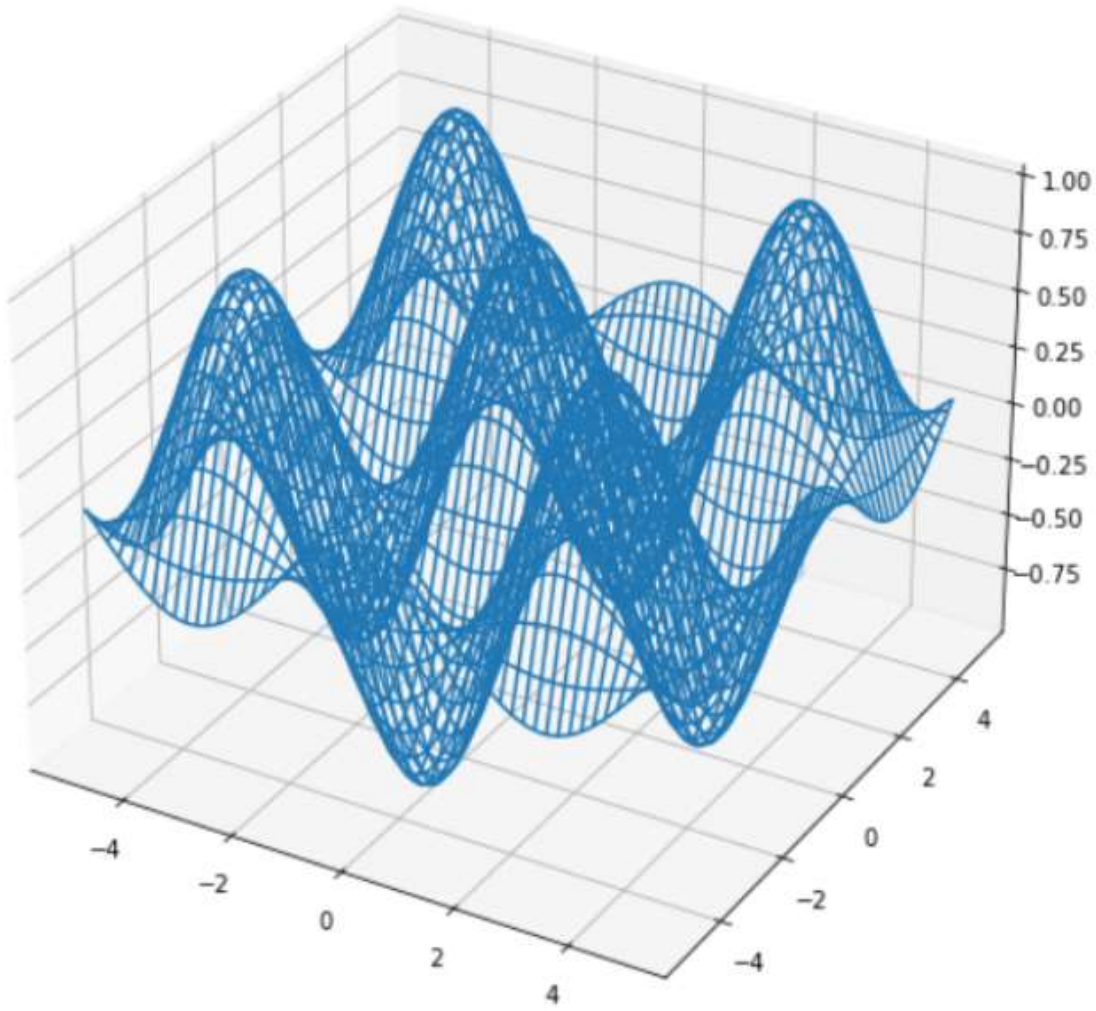
In [ ]:
```
fig = mpl.figure(figsize=(14,9))
# add the code
```

3D Plots

Matplotlib can also generate plots in 3D. To enable this functionality we will need to add the `projection='3d` parameter to the `fig.add_subplot` command. There are multiple methods available in Matplotlib to plot data in 3D. In this example however, we will focus on plotting a 3D surface that we will build through the trigonometrical expression $z = cos(x) * cos(y)$.

First, we store a finite range of numbers (between -5 and 5) in `x` and `y`. Then, we need to transform these numerical ranges into coordinates to render a 3D object by using the method `np.meshgrid`. After, we store our trigonometrical expression in `z`, making sure that we use the data we previously transform with `np.meshgrid`. Finally, we use the `ax.plot_wireframe` command, sending the information stored in `x`, `y`, and `z`.

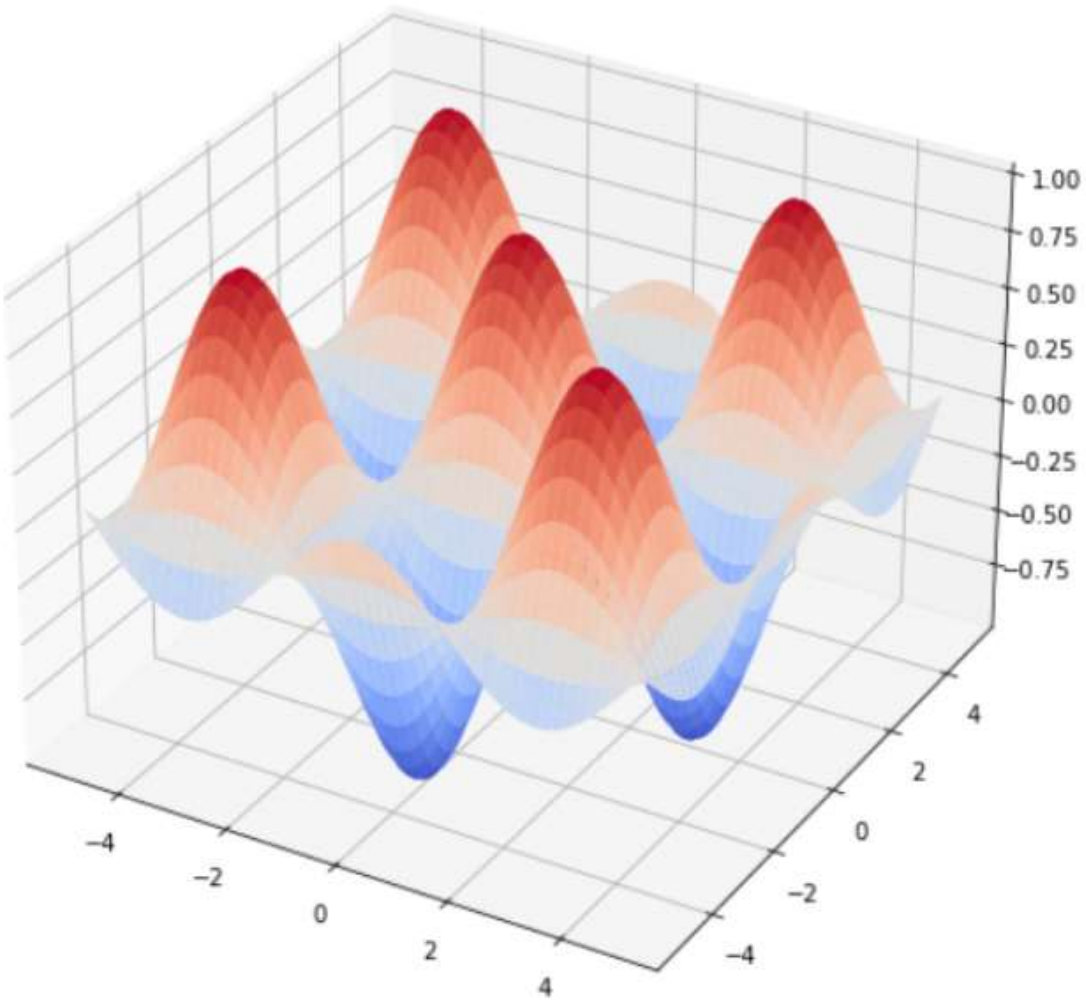
> **Aditional Material**: for a more detailed overview of some of the possibilities offered by `projection='3d'`, see the results from the official Matplotlib documentation.



In [ ]:
```
fig = mpl.figure(figsize=(14,9))
ax = fig.add_subplot(projection='3d')

x = np.arange(-5,5,0.05)
y = np.arange(-5,5,0.05)
# add the code here
```

As a final step, we can further improve the appearance of the 3D surface. For this, we will make three additional changes. First, we are going to change our plotting command to `ax.plot_surface`, so the 3D surface is rendered as a solid instead of a grid. Second, we enable the use of the built-in color maps in Matplotlib, by adding the parameter `cmap=mpl.cm.coolwarm` to the `ax.plot_surface` command. And third, we add the `antialiased=True` parameter to the plotting command, so smooth the edges of the surface.

> **Additional Material**: For a more detailed review of the options offered by `mpl.cm`, review the colormap reference from Matplolib.



In [ ]:
```
fig = mpl.figure(figsize=(14,9))
ax = fig.add_subplot(projection='3d')

x = np.arange(-5,5,0.05)
y = np.arange(-5,5,0.05)
x,y = np.meshgrid(x,y)

# add the code here
```

This last plot will conclude Lab I. 😴 🎉

Make sure to review all the different options and parameters that each function can offer you to improve your data representation plots! 🥸

> 📚 **Additional Material**: For a complete overview of all the possibilitites Matplotlib has to offer, make sure to review the official example gallery.
