# Complete_Guide_to_Matplotlib

June 4, 2025

**Matplotlib**

**What is Matplotlib?**

**Matplotlib** is a powerful and widely used **data visualization library** in Python. It helps create static, animated, and interactive plots. Think of it as the **foundation of all other Python plotting libraries** like Seaborn, Pandas plots, and Plotly.

It was created by **John Hunter in 2002** and is now one of the standard tools used by data scientists, analysts, and engineers for visualizing data.

---

**Where is Matplotlib Used?**

Matplotlib is commonly used in:

- **Data analysis and exploration**
- **Data storytelling and dashboards**
- **Scientific computing and research**
- **Machine learning (model evaluation)**
- **Engineering and simulations**
- **Education** (to teach graphing, math, or stats)

It's especially useful when working in:

- **Jupyter Notebooks**
- **Python scripts**
- **Interactive development environments (IDEs)** like VSCode or PyCharm

---

**Types of Visualizations You Can Make with Matplotlib**

Here are the most common types:

| Visualization Type | Purpose |
|---|---|
| **Line Plot** | Show trends over time or sequences |
| **Scatter Plot** | Show relationships or clustering |
| **Histogram** | Show frequency distribution |
| **Bar Chart** | Compare quantities (vertical bars) |
| **Horizontal Bar Chart** | Same as above, but horizontal |
| **Error Bar Chart** | Show variability or confidence intervals |
| **Stacked Bar Chart** | Compare totals and their parts |

| Visualization Type | Purpose |
|---|---|
| **Pie Chart** | Show part-to-whole relationships |
| **Box Plot** | Show distributions and outliers |
| **Area Chart** | Like line plots but filled |
| **Contour Plot** | Show 3D data in 2D using curves/levels |
| **Heatmap (via `imshow`)** | Show intensity data like correlation matrices |
| **Stem Plot** | Discrete data in a continuous plot |
| **3D Plots** | Using `mpl_toolkits.mplot3d` for surface/lines |

You can also **customize** plots heavily: colors, styles, annotations, legends, titles, axes, grid, background, etc.

---

**Drawbacks / Limitations of Matplotlib**

1. **Verbosity**
   - You often need many lines of code for a simple plot (especially in object-oriented style).
2. **Old Default Styling**
   - Older versions didn't look very modern by default (though styles have improved since v2.0).
3. **Limited Interactivity**
   - Static plots by default. You need to use `%matplotlib notebook`, widgets, or external tools for full interactivity.
4. **Not Ideal for Dashboards**
   - It's not the best choice for web dashboards. Use **Plotly**, **Bokeh**, or **Altair** instead.
5. **Slower for Big Data**
   - Rendering very large datasets can be slow.
6. **Steep Learning Curve**
   - The object-oriented approach can be hard for beginners.

---

**Summary**

- **Matplotlib** is the most foundational and versatile plotting tool in Python.
- Use it for **any static visualization task**.
- Mastering it makes learning other tools like Seaborn and Plotly much easier.
- Combine it with **NumPy**, **Pandas**, and **Jupyter** for full power.

## 1 Table of Contents

## 2 Introduction

When we want to explain something to others, it's much easier to use **pictures, charts, or graphs** instead of just numbers or text. This method is called **Data Visualization**.

Data visualization takes numbers and turns them into visual formats like **charts, graphs, and tables**, so we can easily understand and find patterns in the data. It helps us make better and faster decisions by making complex information **clear and simple**.

In this project, we will focus on **Matplotlib**, a basic and powerful tool in Python used for creating visualizations. Python also has many other tools for visualizing data. First, we'll take a quick look at them, and then we'll learn about Matplotlib in detail.

## 3 Overview of Python Visualization Tools

Python is a popular language used by **data scientists**, and one big reason is that it has **many tools** for creating charts and graphs.

These tools help us **see and understand data more easily**.

Here are some of the main Python visualization tools:

- **Matplotlib** – The most basic and powerful tool for plotting
- **Seaborn** – Built on Matplotlib, used for better-looking statistical plots
- **Pandas** – Has simple built-in plotting features
- **Bokeh** – Used for making interactive web plots
- **Plotly** – Another tool for creating interactive plots (also used in R)
- **ggplot** – Based on R's popular ggplot2 system
- **Pygal** – Good for interactive SVG (vector) charts

In this project, we will focus only on **Matplotlib**, the foundation of most other Python plotting libraries.

## 4 Introduction to Matplotlib

**Matplotlib** is the most basic and widely used library for creating plots and charts in Python.

It is powerful and can do many things:

- Create simple and complex graphs
- Export charts in many formats like **PDF, PNG, JPG, SVG, BMP, and GIF**
- Make many types of visualizations like **line plots, scatter plots, bar charts, histograms, pie charts, box plots**, and even **3D plots**

Many other libraries like **pandas** and **Seaborn** are actually built on top of Matplotlib — they just make it easier to use by writing less code.

**Fun fact:** Matplotlib was created in **2002 by John Hunter**. He originally made it to help visualize brain data (ECoG data) for epilepsy research. Later, it became the **most popular plotting tool** in Python and was even used in NASA's **Phoenix spacecraft mission** in 2008!

## 5 Importing Matplotlib

Before, we need to actually start using Matplotlib, we need to import it. We can import Matplotlib as follows:-

**import matplotlib**

Most of the time, we have to work with pyplot interface of Matplotlib. So, I will import pyplot interface of Matplotlib as follows:-

**import matplotlib.pyplot**

To make things even simpler, we will use standard shorthand for Matplotlib imports as follows:-

**import matplotlib.pyplot as plt**

[1]:
```python
# Import dependencies

import numpy as np
import pandas as pd
```

[2]:
```python
# Import Matplotlib
import matplotlib.pyplot as plt
```

## 6 Displaying Plots in Matplotlib (Simplified)

How and **where your plot appears** depends on the **environment** you're using Matplotlib in. There are **three main situations**, and each has its own way to display plots.

---

1. **Plotting from a Script (like .py file)**

- Use `plt.show()` at the end of your script.
- It opens a window that displays your plot.
- You should use `plt.show()` **only once** at the end — using it multiple times may cause errors or unexpected behavior.

---

2. **Plotting from an IPython Shell**

- This is like working from a terminal or command line using IPython.
- Use the magic command `%matplotlib` to turn on interactive mode.
- Then, any plot you create will open in a window and can be updated interactively.

---

3. **Plotting from a Jupyter Notebook**

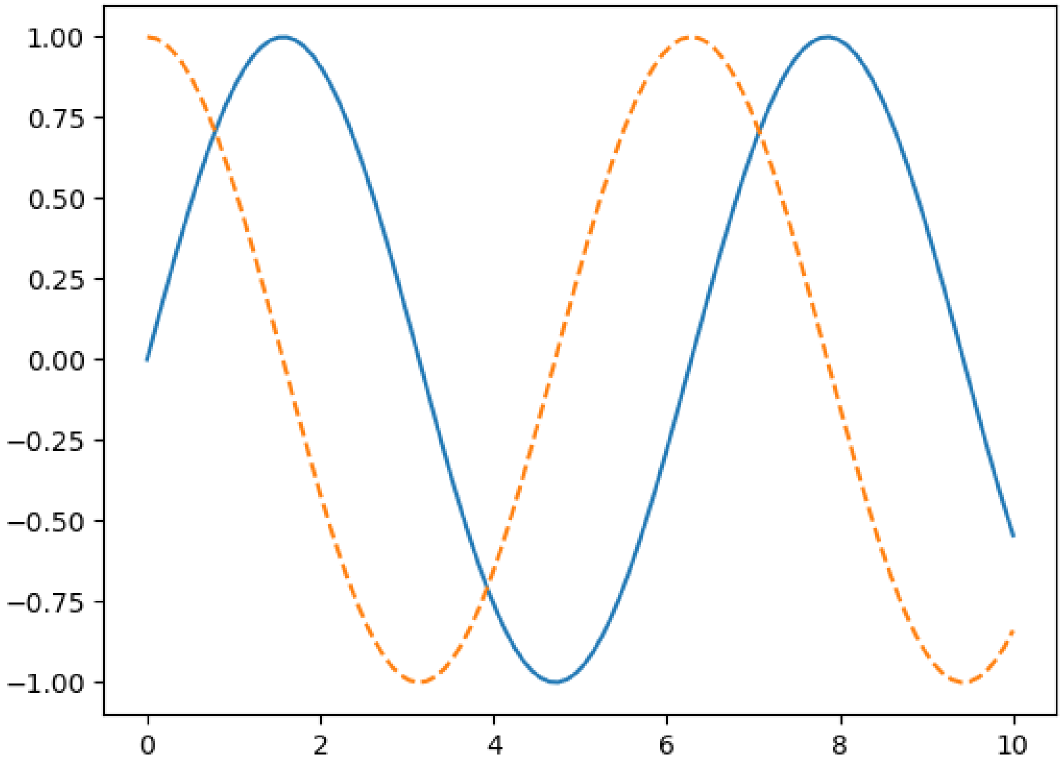
- Jupyter Notebooks are popular for data analysis and learning.
- To display plots inside the notebook, use one of these commands **at the top**:
  - `%matplotlib inline` → Shows plots as **static images** below your code cells.

  – `%matplotlib notebook` → Shows **interactive plots** right inside the notebook (you can zoom, move, etc.).

You only need to use the `%matplotlib` command **once per notebook session**.

After that, any `plt.plot()` you run will show the graph right below the code cell.

[9]:
```
%matplotlib inline

x1 = np.linspace(0, 10, 100)

# create a plot figure
fig = plt.figure()

plt.plot(x1, np.sin(x1), '-')
plt.plot(x1, np.cos(x1), '--')
plt.show()
```



## 7 Matplotlib Object Hierarchy

Matplotlib plots are made up of different objects arranged like a tree.

- The **Figure** is the outer container — like a big box that holds everything.

- Inside the Figure, there are one or more **Axes**. Each Axes is a single plot or graph.
- Each Axes contains smaller parts like lines, tick marks, labels, legends, and titles.

Think of it like this: **Figure = a big box Axes = smaller boxes inside the big box where the actual plots are drawn Other parts (ticks, lines, labels) = things inside the smaller boxes**

## 8 Matplotlib API Overview

Matplotlib has two main ways to create plots:

1. **Pyplot interface:**
    - This is like MATLAB's style.
    - It uses simple commands that remember what you last did (state-based).
    - Easy for quick plots and beginners.
2. **Object-Oriented (OO) interface:**
    - More powerful and flexible.
    - You create and control Figure and Axes objects directly.
    - Better for complex or multiple plots.

There is also a third way called **pylab**, but it mixes things too much and is not recommended anymore.

## 9 Pyplot API

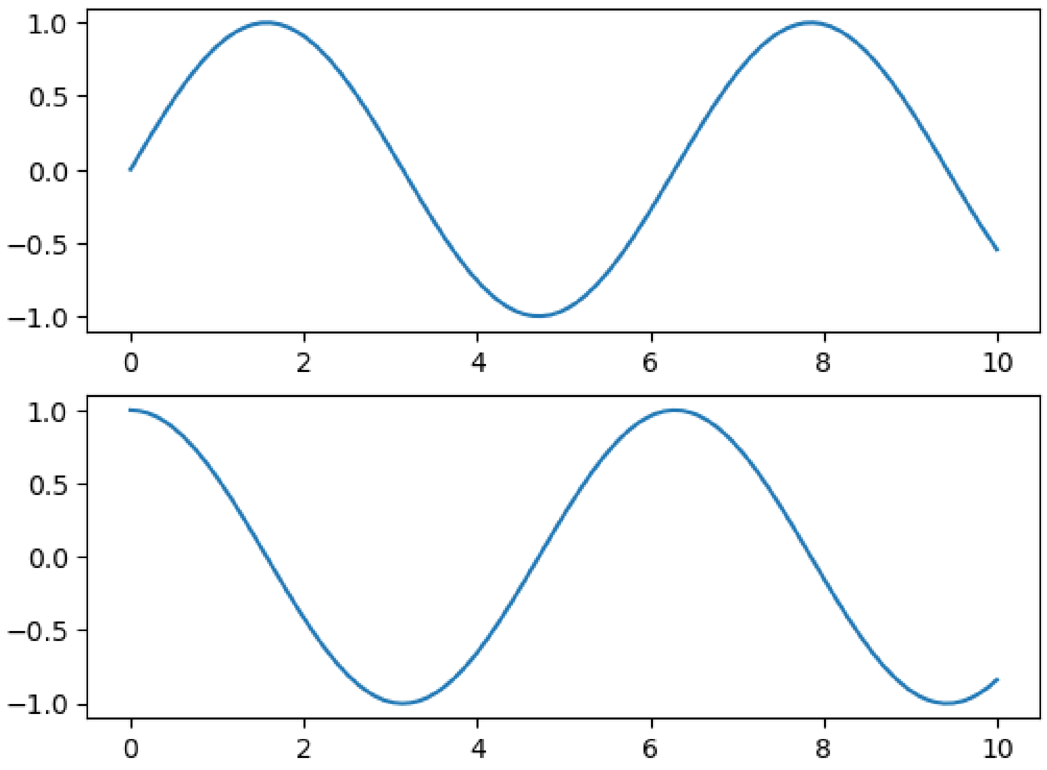
- **Matplotlib.pyplot** (often imported as `plt`) is a set of simple commands that work like MATLAB for plotting.
- It lets you create and change plots step-by-step without dealing with objects directly.
- This is called a **stateful** interface because it remembers the current figure and plot you are working on.
- You don't need to keep track of figure or axes objects yourself — the commands just affect the "current" plot.
- You can get the current figure or axes if needed by:

  ```
  plt.gcf()   # Get current figure
  plt.gca()   # Get current axes
  ```

- This interface is great for quick and interactive plotting because you can run one command and see the result right away.
- But for more complex or customized plots, the **Object-Oriented interface** is better.

[12]:
```
# create a plot figure
plt.figure()
```

```
# create the first of two panels and set current axis
plt.subplot(2, 1, 1)   # (rows, columns, panel number)
plt.plot(x1, np.sin(x1))

# create the second of two panels and set current axis
plt.subplot(2, 1, 2)   # (rows, columns, panel number)
plt.plot(x1, np.cos(x1));
plt.show()
```

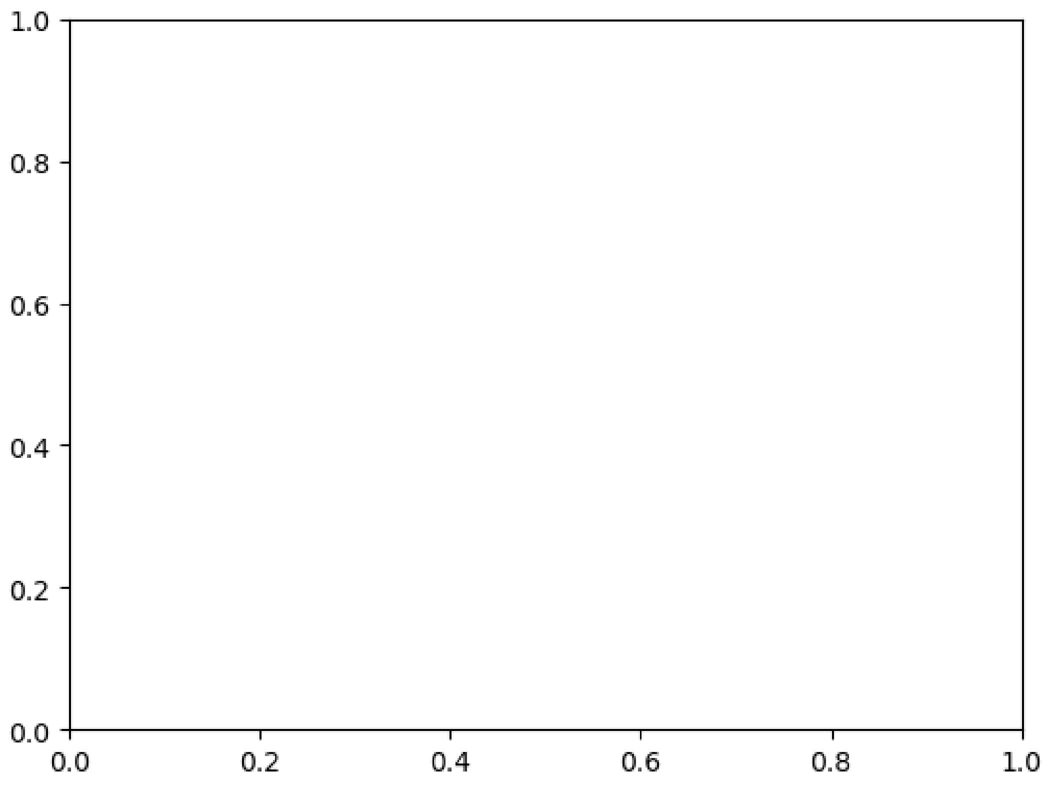


[13]:
```
# get current figure information

print(plt.gcf())
```

```
Figure(640x480)
```

[15]:
```
# get current axis information

print(plt.gca())
plt.show()
```

```
Axes(0.125,0.11;0.775x0.77)
```

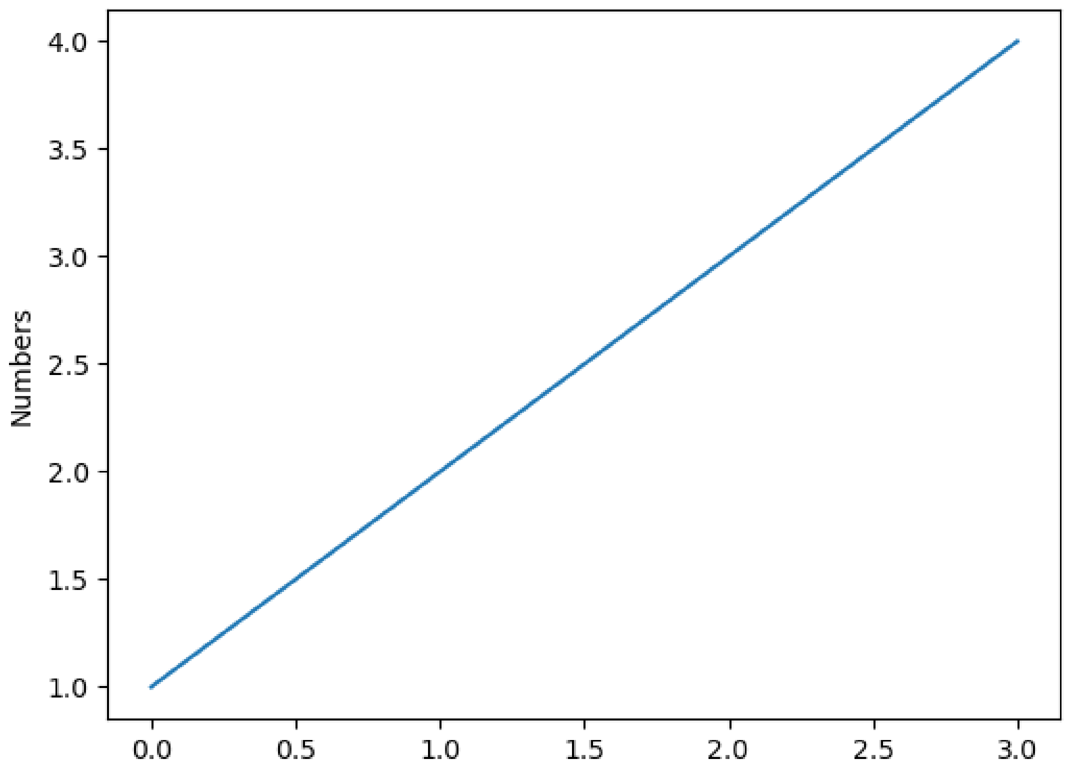


**Visualization with Pyplot**

- Making plots with Pyplot is very easy.
- Suppose your y-values are `[1, 2, 3, 4]`.
- If you only give these y-values to `plt.plot()`, Matplotlib will automatically create x-values starting from 0.
- So the x-values will be `[0, 1, 2, 3]` by default.
- This means Matplotlib plots points: (0,1), (1,2), (2,3), and (3,4).

In short: If you don't provide x-values, Pyplot uses `[0, 1, 2, ...]` to match the length of your y-values.

[16]:
```
plt.plot([1, 2, 3, 4])
plt.ylabel('Numbers')
plt.show()
```

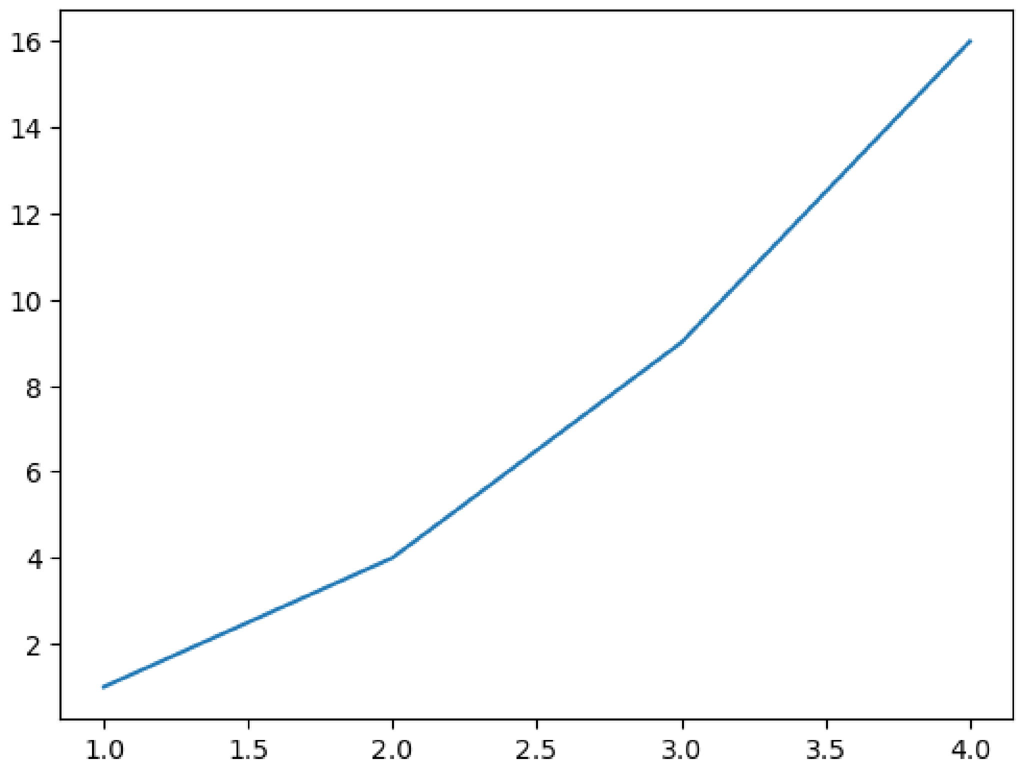


**plot() - A versatile command**

plot() is a versatile command. It will take an arbitrary number of arguments.

For example, to plot x versus y, we can issue the following command:-

[17]:
```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
plt.show()
```

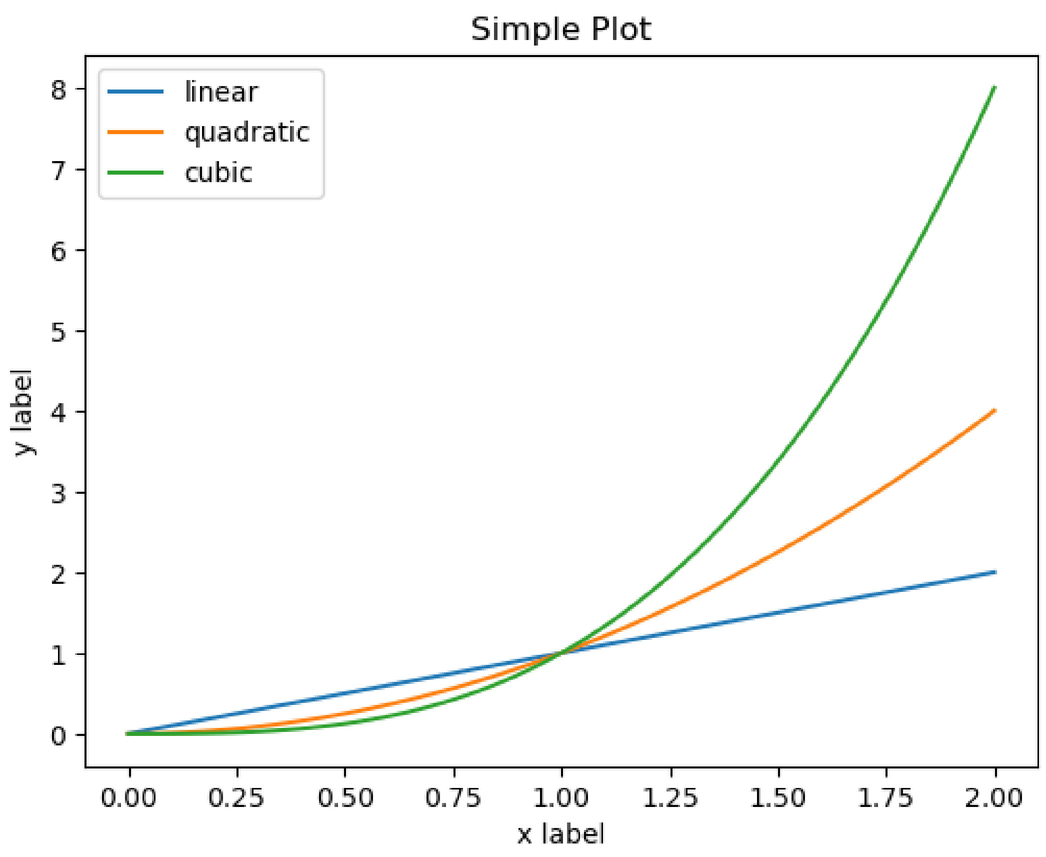


**State-machine interface**

Pyplot provides the state-machine interface to the underlying object-oriented plotting library. The state-machine implicitly and automatically creates figures and axes to achieve the desired plot.

For example:

[18]:
```
x = np.linspace(0, 2, 100)

plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')

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

plt.title("Simple Plot")

plt.legend()

plt.show()
```

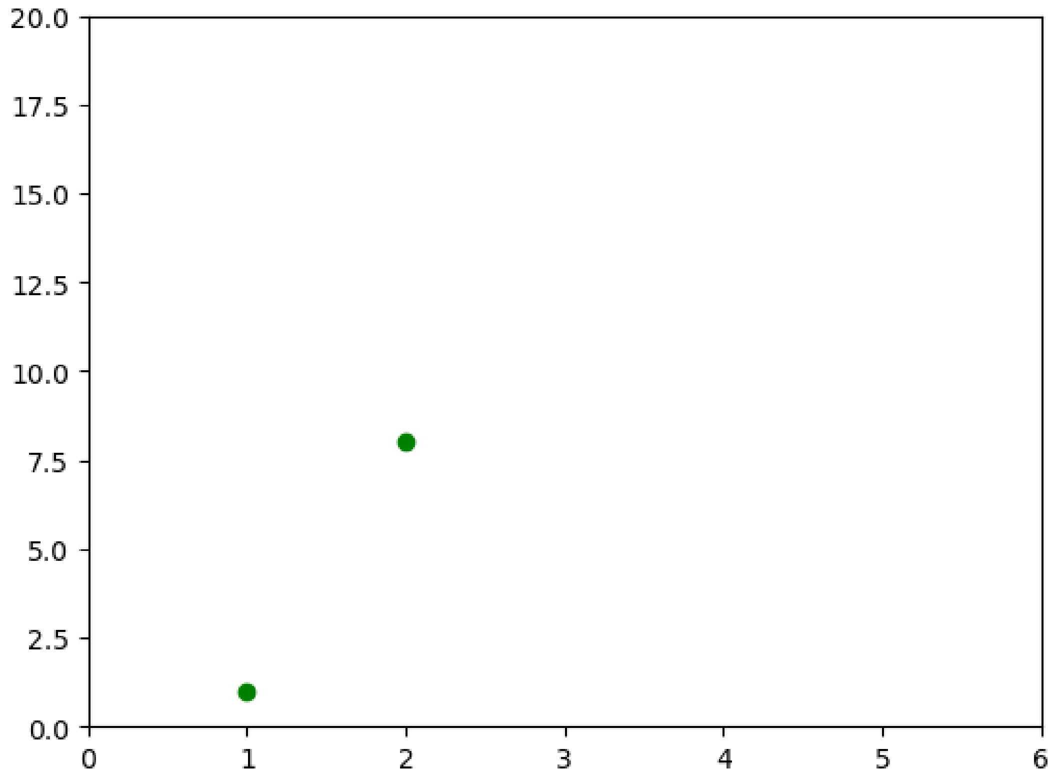


**Formatting the Style of a Plot**

- When you plot with `plt.plot(x, y)`, you can add a **third argument** called the **format string**.
- This string tells Matplotlib the **color** and **line style** to use.
- The format string uses letters and symbols similar to MATLAB.
- For example:
  - `'b-'` means a **blue solid line** (this is the default).
  - `'ro'` means **red circles** (no lines, just points).
  - You can combine a color and line style, like `'g--'` for a **green dashed line**.

  For example, to plot the above line with red circles, we would issue the following command:-

[19]:
```
plt.plot([1, 2, 3, 4], [1, 8, 27, 64], 'go')
plt.axis([0, 6, 0, 20])
plt.show()
```

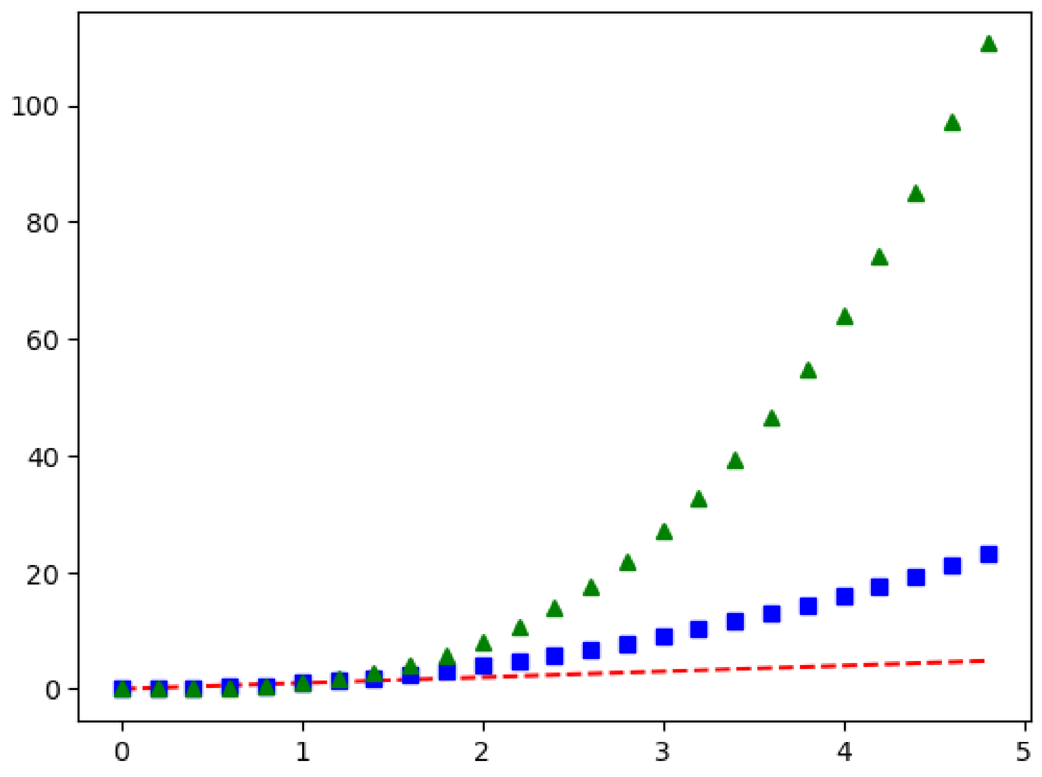


The axis() command in the example above takes a list of [xmin, xmax, ymin, ymax] and specifies the viewport of the axes.

**Working with NumPy arrays**

Generally, we have to work with NumPy arrays. All sequences are converted to numpy arrays internally. The below example illustrates plotting several lines with different format styles in one command using arrays.

[20]:
```
# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```

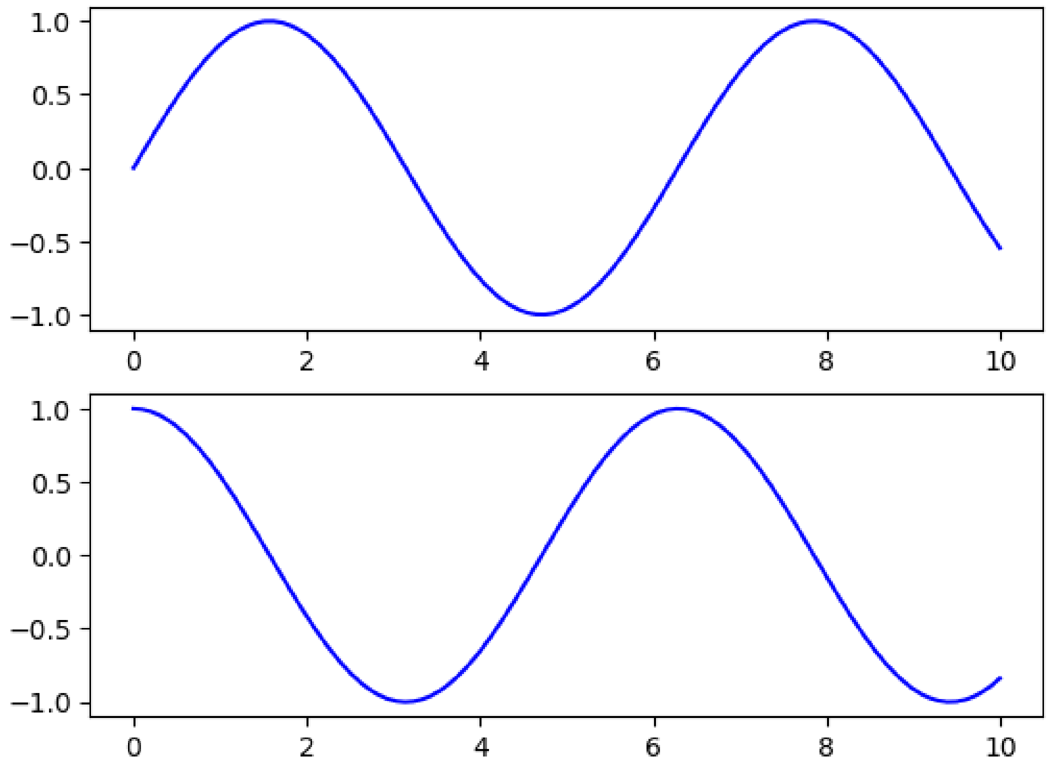


## 10 Object-Oriented API

- The **Object-Oriented (OO) API** is used for more complex or customized plots.
- Unlike Pyplot, where you rely on the "current" figure and axes, here you **create and control Figure and Axes objects directly**.
- This gives you **more control** over every part of your plot.
- Think of it like this:
  - **Figure** = the big container (like a box) that holds everything.
  - **Axes** = smaller containers inside the Figure, each representing a single plot.
  - Each Axes has its own axis, tick marks, lines, legends, titles, etc.
- You call methods on these objects to create and modify the plots.

[23]:
```
# First create a grid of plots
# ax will be an array of two Axes objects
fig, ax = plt.subplots(2)



# Call plot() method on the appropriate object
ax[0].plot(x1, np.sin(x1), 'b-')
```

```
ax[1].plot(x1, np.cos(x1), 'b-')
plt.show()
```



**Objects and Reference**

The main idea with the Object Oriented API is to have objects that one can apply functions and actions on. The real advantage of this approach becomes apparent when more than one figure is created or when a figure contains more than one subplot.

We create a reference to the figure instance in the fig variable. Then, we ceate a new axis instance axes using the add_axes method in the Figure class instance fig as follows:-

[27]:
```
fig = plt.figure()

x2 = np.linspace(0, 5, 10)
y2 = x2 ** 2

axes = fig.add_axes([0.1, 0.1, 0.8, 0.8])

axes.plot(x2, y2, 'r')

axes.set_xlabel('x2')
axes.set_ylabel('y2')
axes.set_title('title')
plt.show()
```

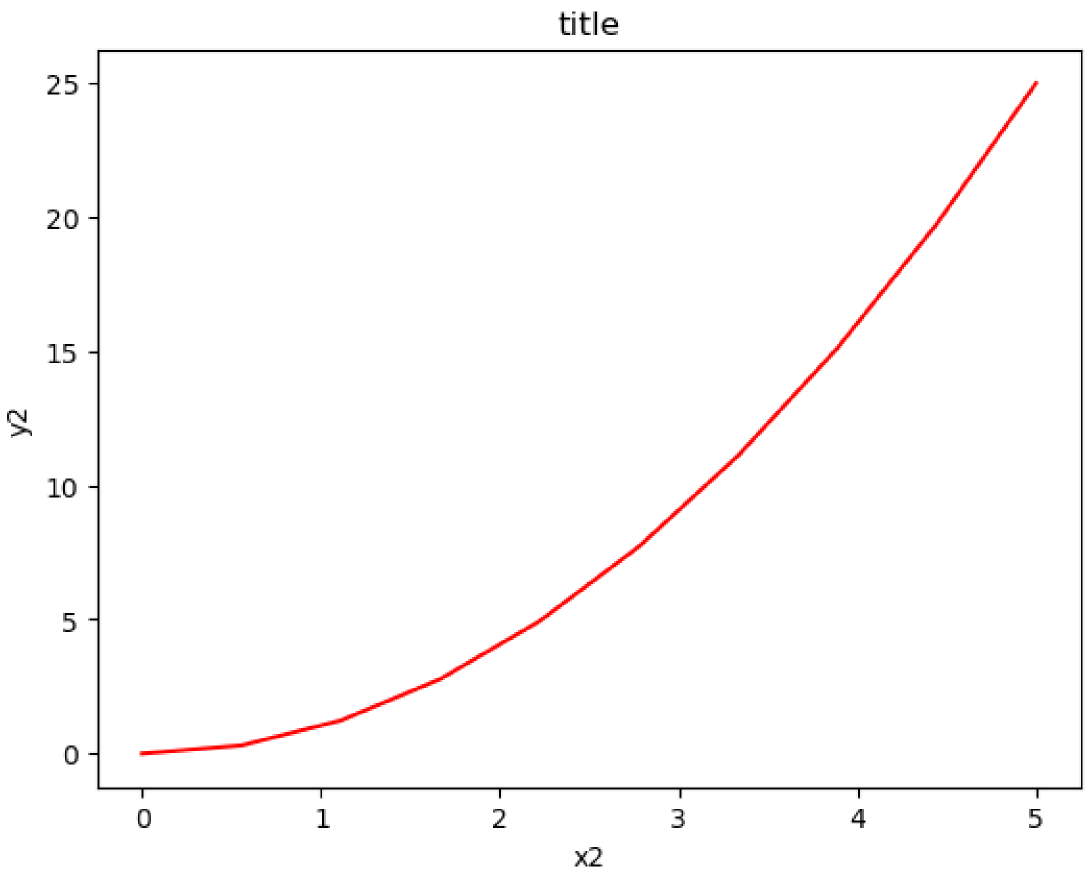


**Figure and Axes**

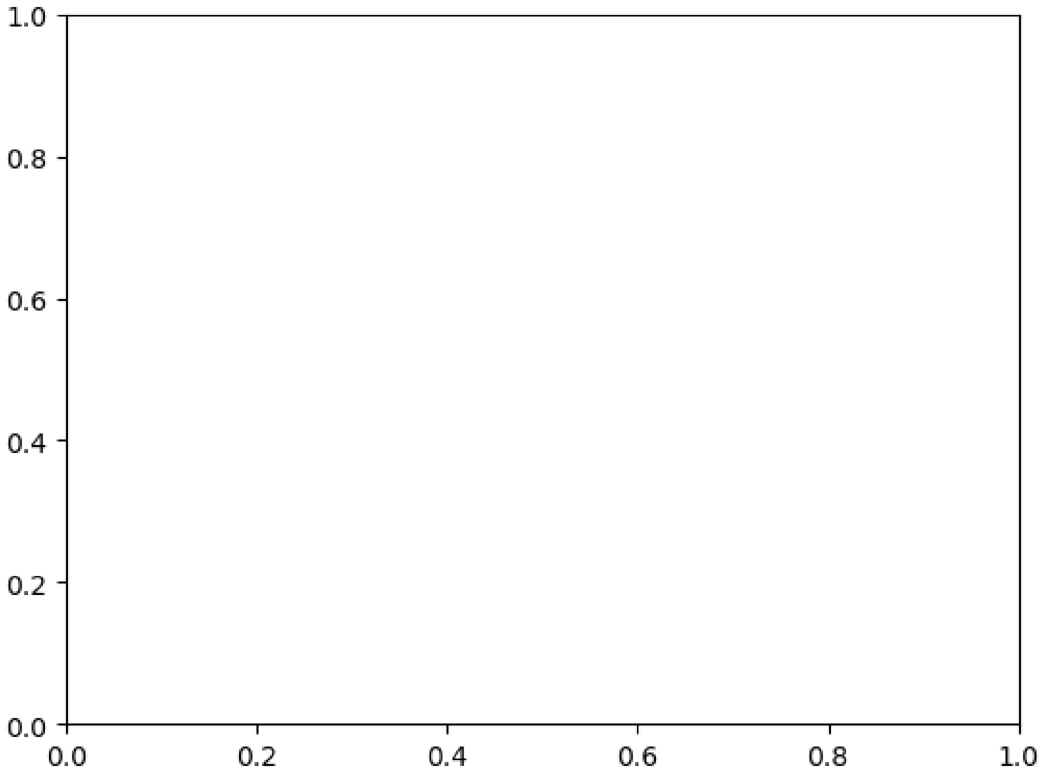
- To start plotting with the Object-Oriented API, you first create a **Figure** and **Axes**.
- You can do this by:

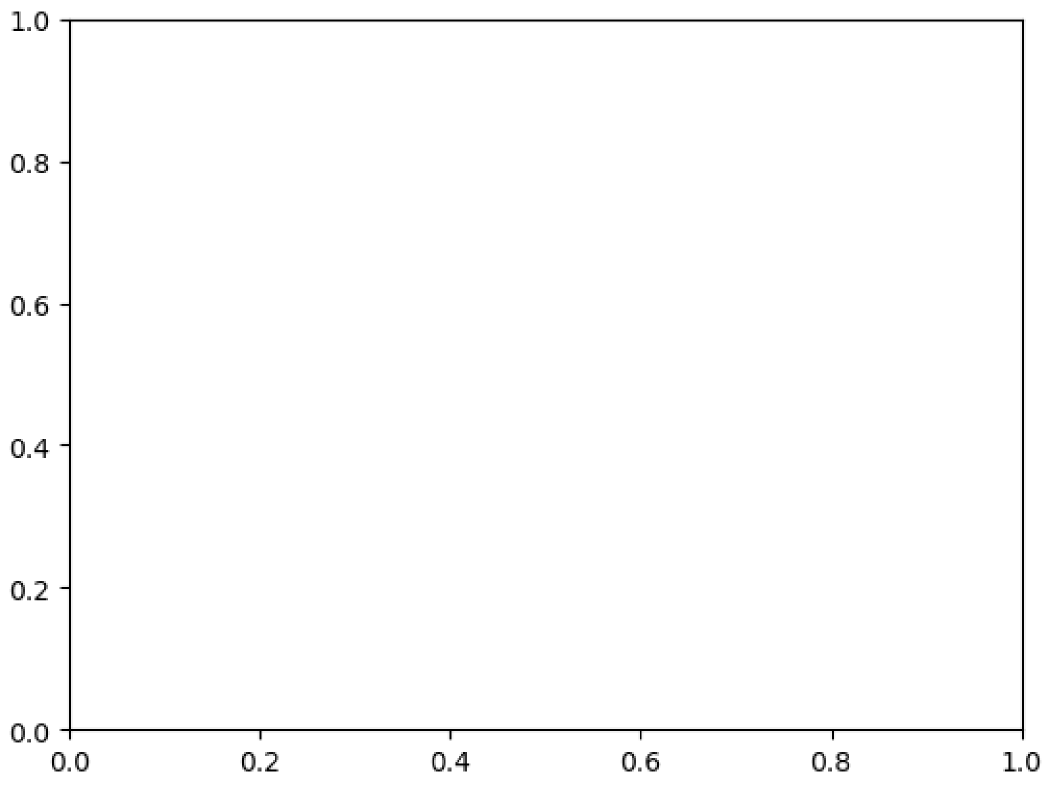
  ```
  fig = plt.figure()   # Create a Figure object (the big container)
  ax = plt.axes()      # Create an Axes object (the plot area inside the Figure)
  ```

- **Figure** (`fig`) is the outer container that holds everything: axes, titles, labels, lines, etc.
- **Axes** (`ax`) is the part where the actual plot happens. It's like a box with ticks, labels, and the data plots.
- We usually name the figure as `fig` and the axes as `ax` by convention.

[29]:
```
fig = plt.figure()

ax = plt.axes()
```

```
plt.show()
```





## 11 Figure and Subplots

- In Matplotlib, all plots live inside a **Figure**.
- You can create a new figure by:

```
fig = plt.figure()
```

- You can add **multiple subplots** (small plots inside one figure) using:

```
ax1 = fig.add_subplot(2, 2, 1)
```

- This means:
  - You want a grid of 2 rows and 2 columns (total 4 subplots).
  - `1` means you are selecting the **first subplot** (numbering starts at 1, going left to right, top to bottom).
- Similarly, you can create the other subplots:

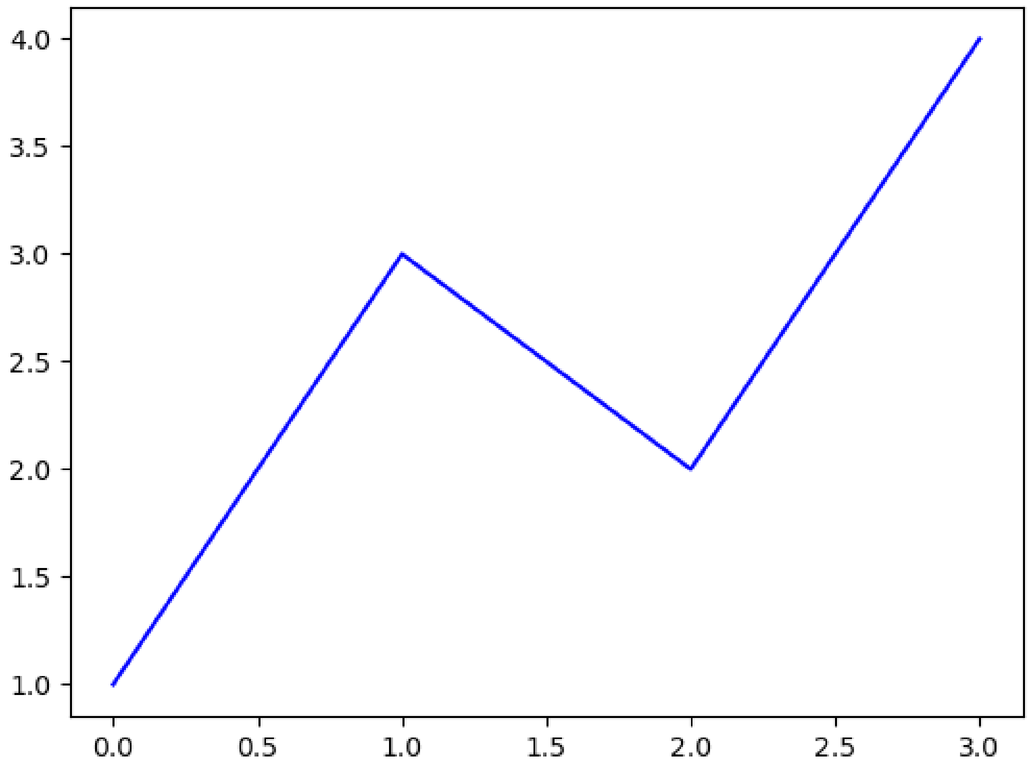
```
ax2 = fig.add_subplot(2, 2, 2)  # second subplot
ax3 = fig.add_subplot(2, 2, 3)  # third subplot
ax4 = fig.add_subplot(2, 2, 4)  # fourth subplot
```

- So, your figure now has 4 smaller plots arranged in a 2x2 grid.

## 12 First plot with Matplotlib

Now, I will start producing plots. Here is the first example:-

```
[33]: plt.plot([1, 3, 2, 4], 'b-')

plt.show( )
```



plt.plot([1, 3, 2, 4], ‘b-’)

This code line is the actual plotting command. Only a list of values has been plotted that represent the vertical coordinates of the points to be plotted. Matplotlib will use an implicit horizontal values list, from 0 (the first value) to N-1 (where N is the number of items in the list).

**Specify both Lists**

Also, we can explicitly specify both the lists as follows:-

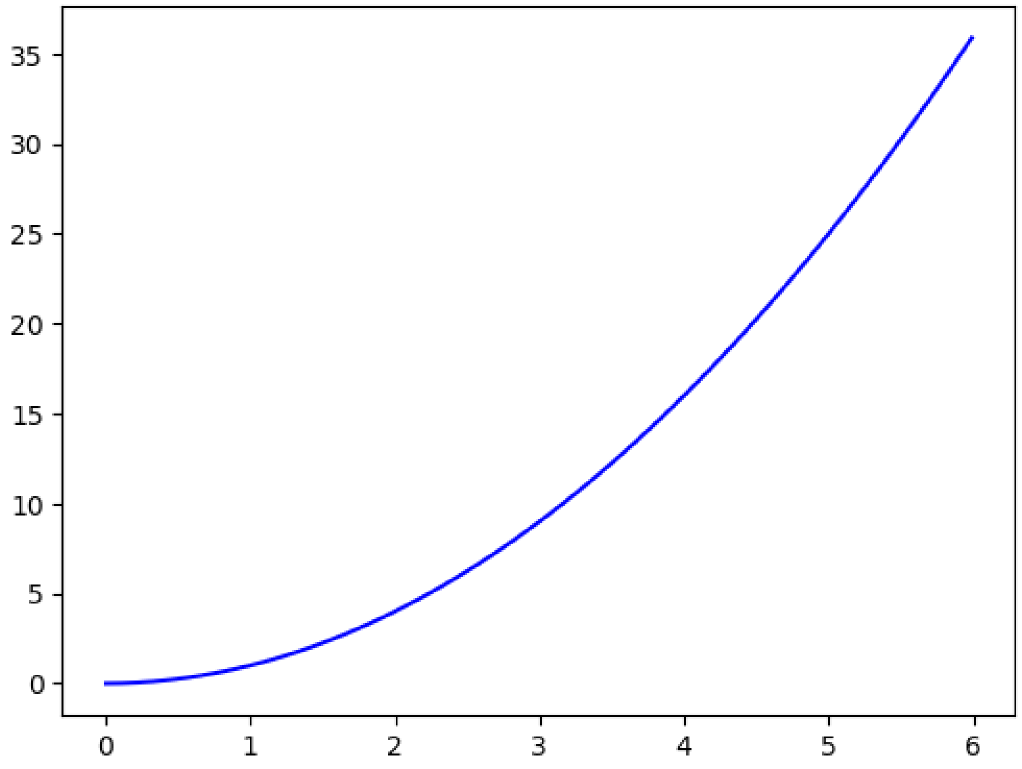
x3 = range(6)

plt.plot(x3, [xi**2 for xi in x3])

plt.show()

```
[34]: x3 = np.arange(0.0, 6.0, 0.01)
```

```
plt.plot(x3, [xi**2 for xi in x3], 'b-')

plt.show()
```

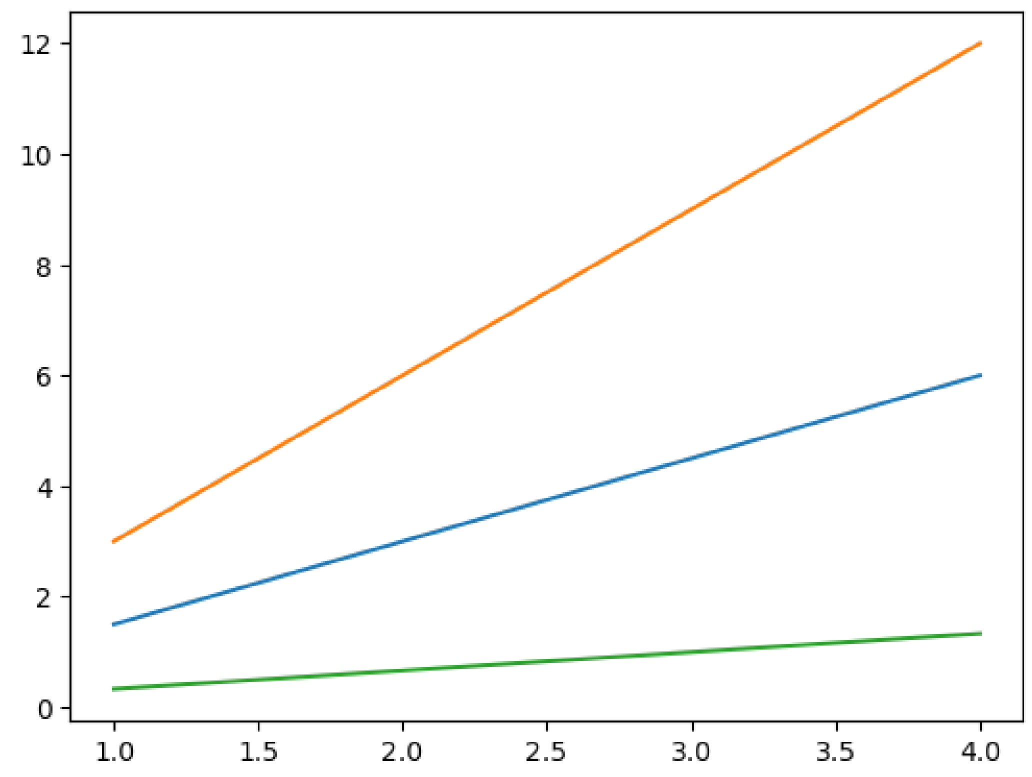


## 13 Multiline Plots

Multiline Plots mean plotting more than one plot on the same figure. We can plot more than one plot on the same figure. It can be achieved by plotting all the lines before calling show().

It can be done as follows:-

[35]:
```
x4 = range(1, 5)

plt.plot(x4, [xi*1.5 for xi in x4])

plt.plot(x4, [xi*3 for xi in x4])

plt.plot(x4, [xi/3.0 for xi in x4])

plt.show()
```



## 14 Parts of a Plot

There are different parts of a plot. These are title, legend, grid, axis and labels etc. These are denoted in the following figure:-

```
[ ]: ![download.png](attachment:40d28a93-cd52-4043-b831-dc208e430d82.png)
```

## 15 Saving the Plot

We can save the figures in a wide variety of formats. We can save them using the savefig() command as follows:-

**fig.savefig('fig1.png')**

We can explore the contents of the file using the IPython Image object.

from IPython.display import Image

**Image('fig1.png')**

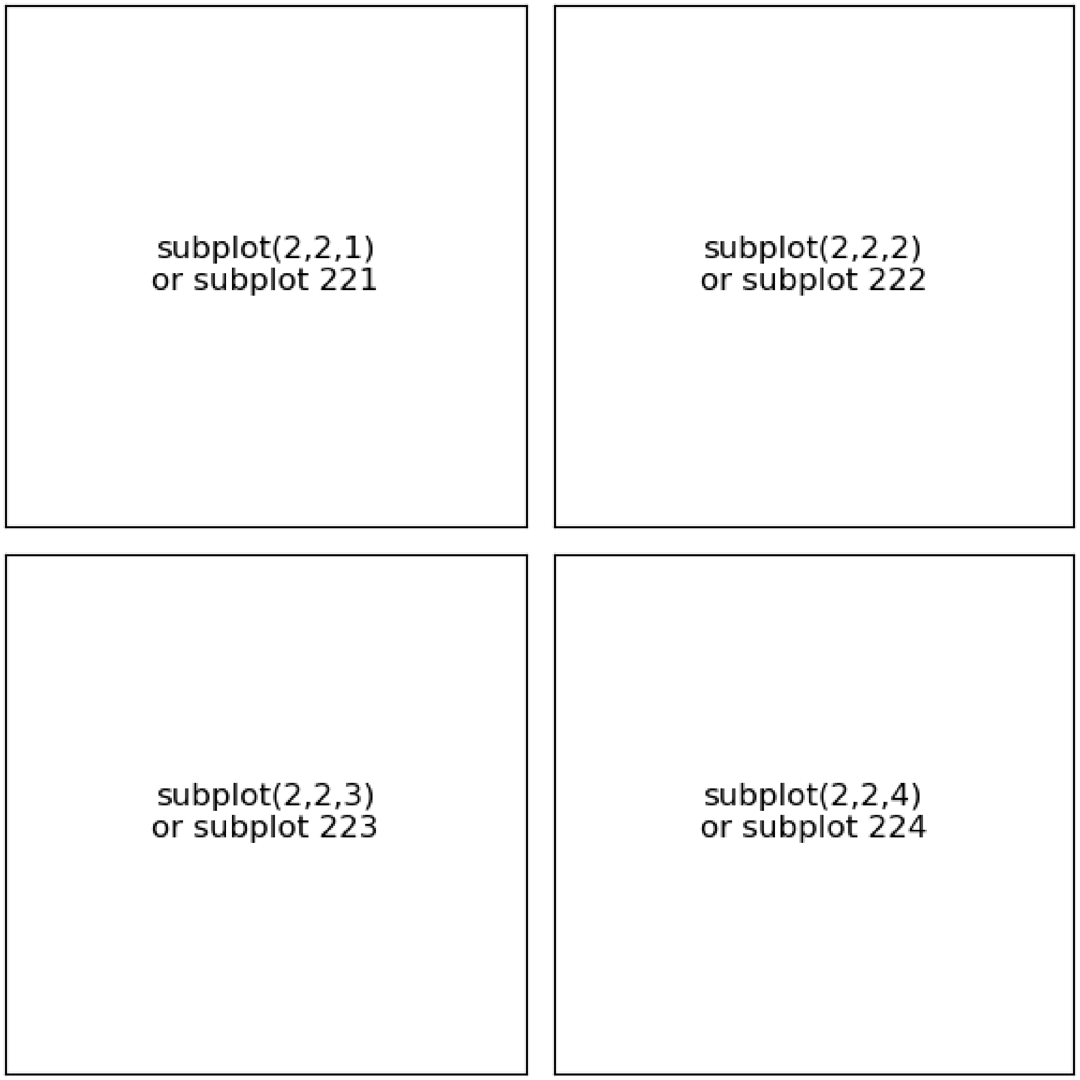
In savefig() command, the file format is inferred from the extension of the given filename. Depending on the backend, many different file formats are available. The list of supported file types can be found by using the get_supported_filetypes() method of the figure canvas object as follows:-

**fig.canvas.get_supported_filetypes()**

[36]:
```
# Saving the figure

fig.savefig('plot1.png')
```

[37]:
```
# Explore the contents of figure

from IPython.display import Image

Image('plot1.png')
```

[37]:



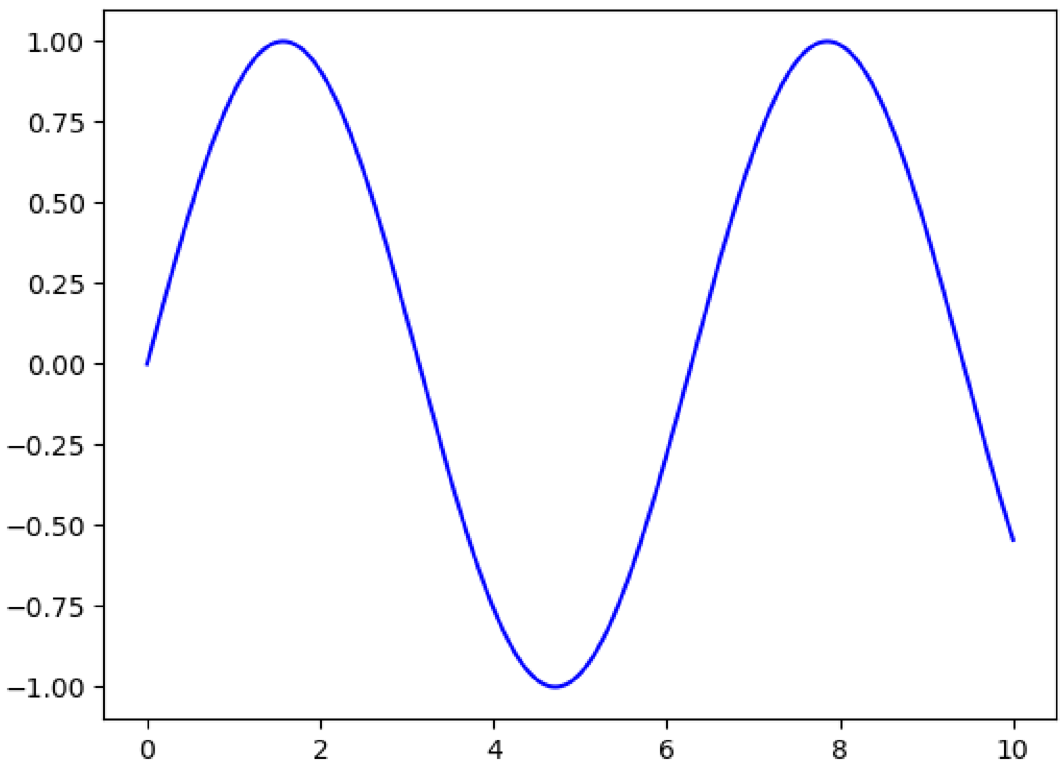
```
[38]: # Explore supported file formats


      fig.canvas.get_supported_filetypes()
```

```
[38]: {'eps': 'Encapsulated Postscript',
       'jpg': 'Joint Photographic Experts Group',
       'jpeg': 'Joint Photographic Experts Group',
       'pdf': 'Portable Document Format',
       'pgf': 'PGF code for LaTeX',
       'png': 'Portable Network Graphics',
       'ps': 'Postscript',
       'raw': 'Raw RGBA bitmap',
       'rgba': 'Raw RGBA bitmap',
       'svg': 'Scalable Vector Graphics',
       'svgz': 'Scalable Vector Graphics',
       'tif': 'Tagged Image File Format',
       'tiff': 'Tagged Image File Format',
       'webp': 'WebP Image Format'}
```

## 16 Line Plot

We can use the following commands to draw the simple sinusoid line plot:-

```
[42]: # Create figure and axes first
      fig = plt.figure()

      ax = plt.axes()

      # Declare a variable x5
      x5 = np.linspace(0, 10, 1000)


      # Plot the sinusoid function
      ax.plot(x5, np.sin(x5), 'b-')
      plt.show()
```



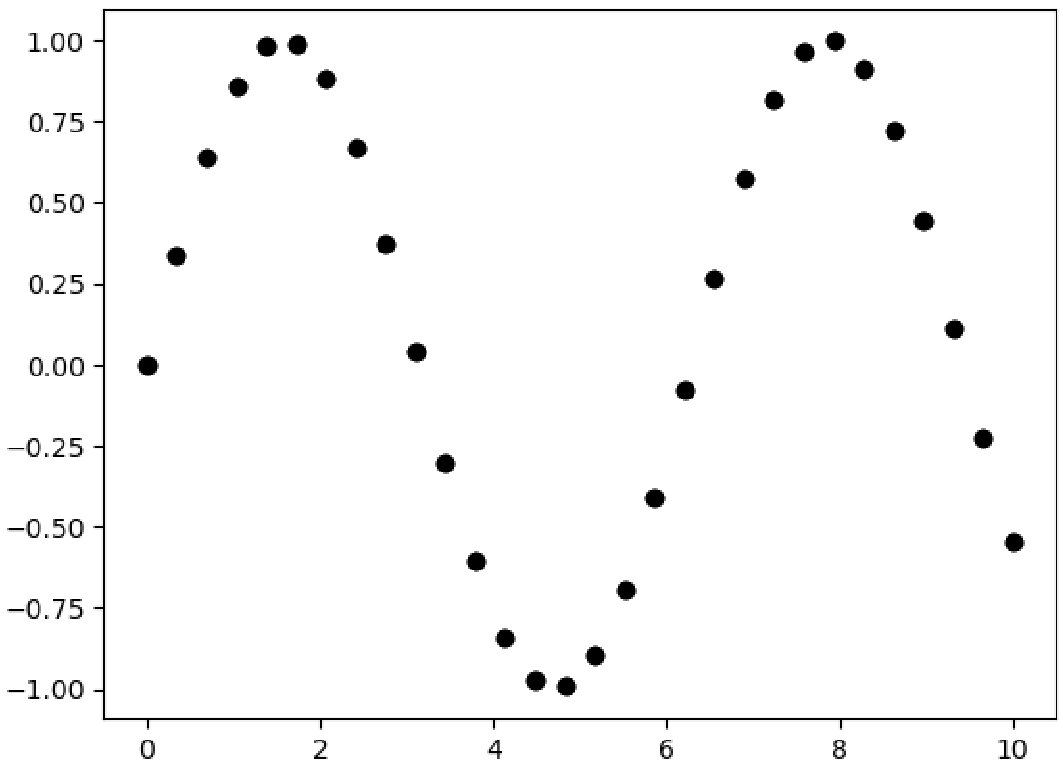
## 17 Scatter Plot

Another commonly used plot type is the scatter plot. Here the points are represented individually with a dot or a circle.

**Scatter Plot with plt.plot()**

We have used plt.plot/ax.plot to produce line plots. We can use the same functions to produce the scatter plots as follows:-

[44]:
```
x7 = np.linspace(0, 10, 30)

y7 = np.sin(x7)

plt.plot(x7, y7, 'o', color = 'black')

plt.show()
```

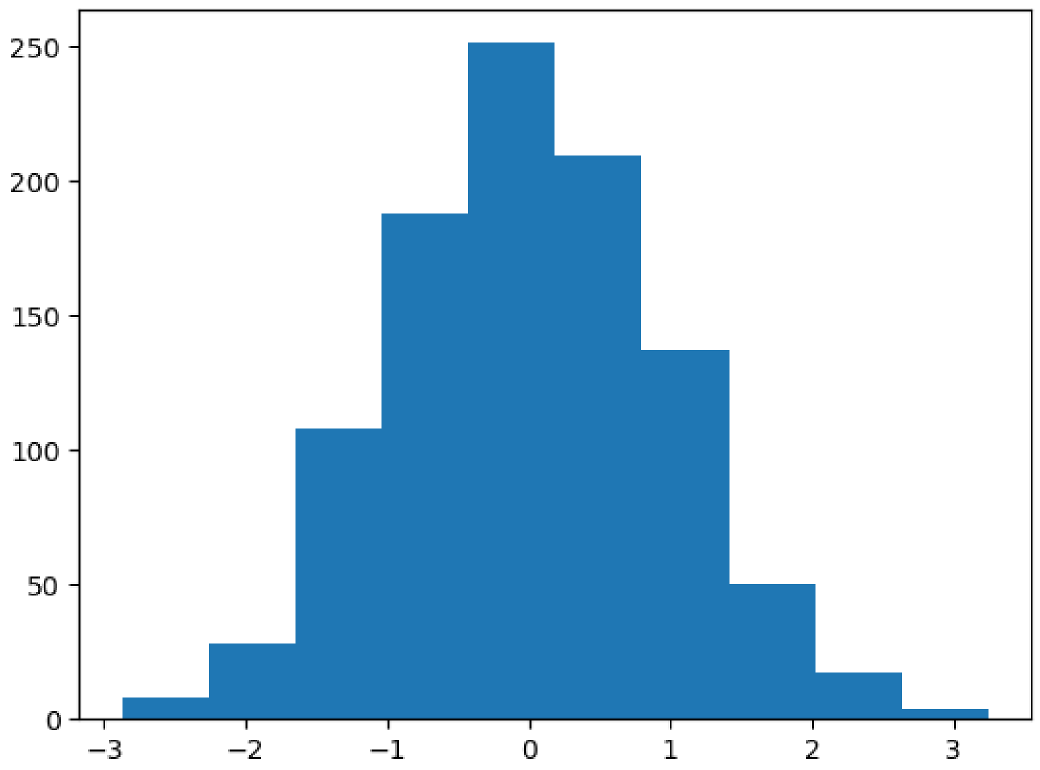


## 18 Histogram

Histogram charts are a graphical display of frequencies. They are represented as bars. They show what portion of the dataset falls into each category, usually specified as non-overlapping intervals. These categories are called bins.

The plt.hist() function can be used to plot a simple histogram as follows:-

[46]:
```
data1 = np.random.randn(1000)

plt.hist(data1)
plt.show()
```

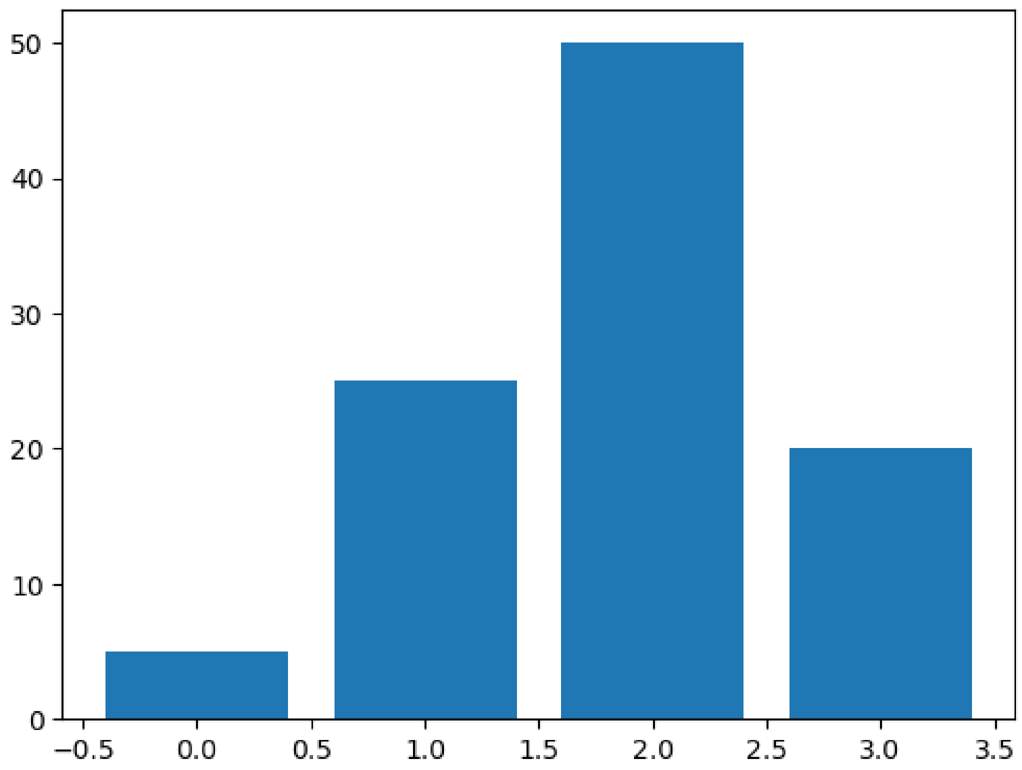


## 19 Bar Chart

Bar charts display rectangular bars either in vertical or horizontal form. Their length is proportional to the values they represent. They are used to compare two or more values.

We can plot a bar chart using plt.bar() function. We can plot a bar chart as follows:-

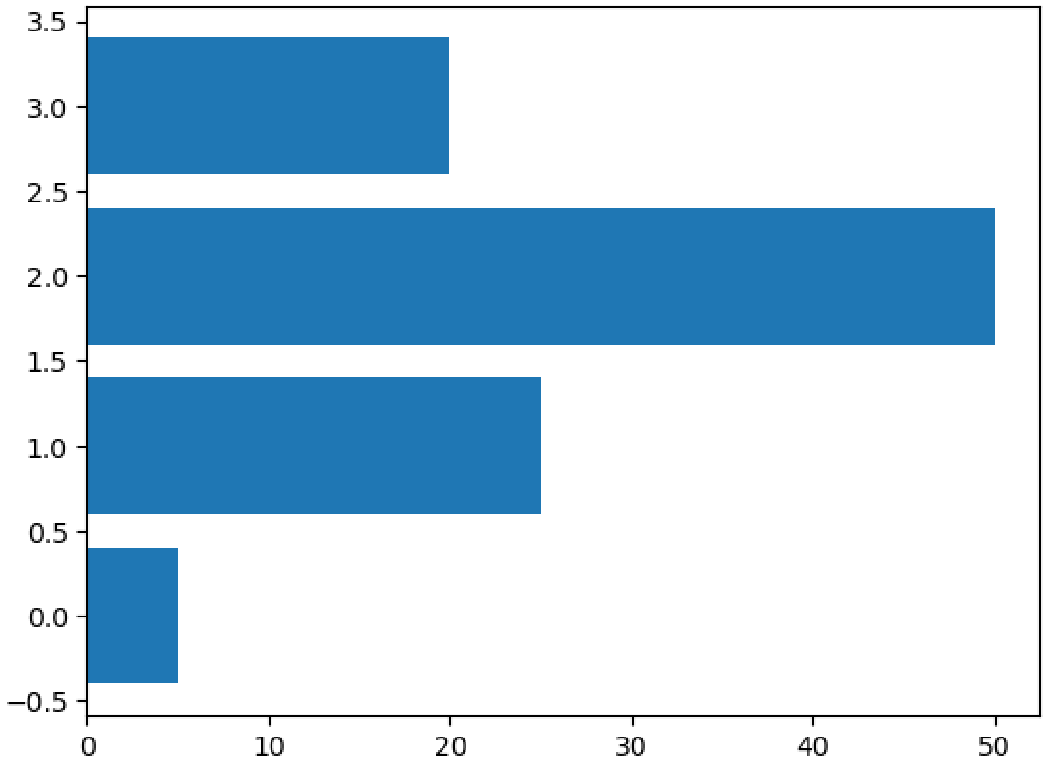
[47]:
```
data2 = [5. , 25. , 50. , 20.]

plt.bar(range(len(data2)), data2)

plt.show()
```



## 20 Horizontal Bar Chart

We can produce Horizontal Bar Chart using the plt.barh() function. It is the strict equivalent of plt.bar() function.

[48]:
```
data2 = [5. , 25. , 50. , 20.]

plt.barh(range(len(data2)), data2)

plt.show()
```

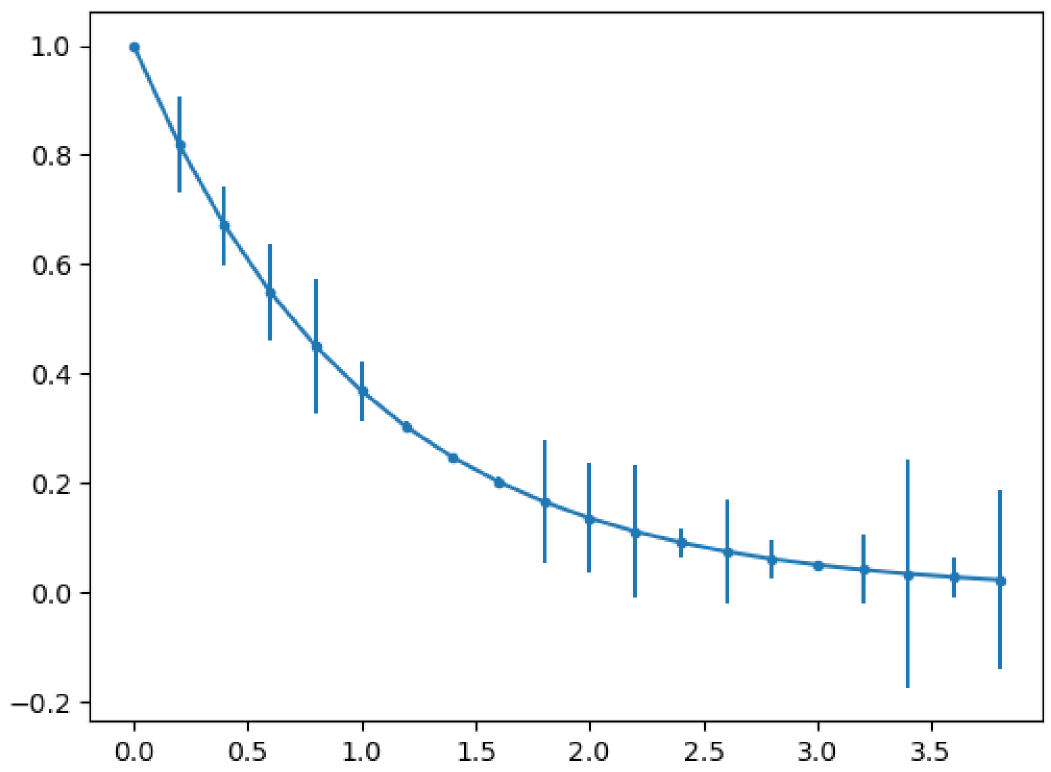


## 21 Error Bar Chart

In experimental design, the measurements lack perfect precision. So, we have to repeat the measurements. It results in obtaining a set of values. The representation of the distribution of data values is done by plotting a single data point (known as mean value of dataset) and an error bar to represent the overall distribution of data.

We can use Matplotlib's errorbar() function to represent the distribution of data values. It can be done as follows:-

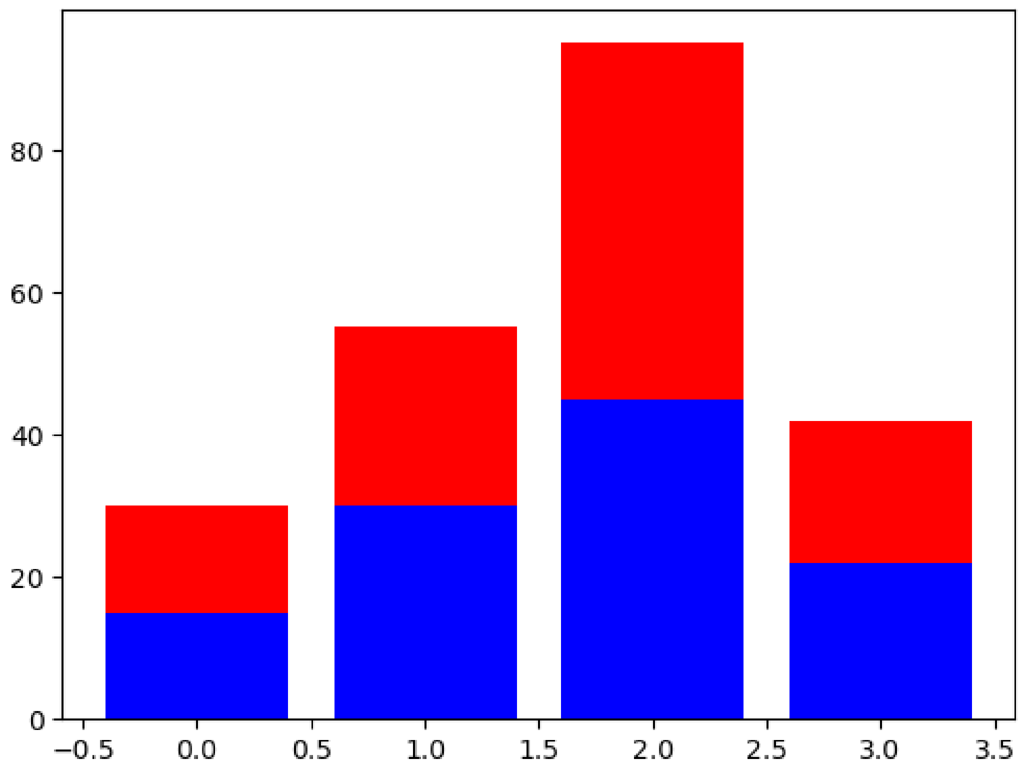
[49]:
```
x9 = np.arange(0, 4, 0.2)

y9 = np.exp(-x9)

e1 = 0.1 * np.abs(np.random.randn(len(y9)))

plt.errorbar(x9, y9, yerr = e1, fmt = '.-')

plt.show();
```



## 22 Stacked Bar Chart

We can draw stacked bar chart by using a special parameter called bottom from the plt.bar() function. It can be done as follows:-

[50]:
```
A = [15., 30., 45., 22.]

B = [15., 25., 50., 20.]

z2 = range(4)

plt.bar(z2, A, color = 'b')
plt.bar(z2, B, color = 'r', bottom = A)

plt.show()
```



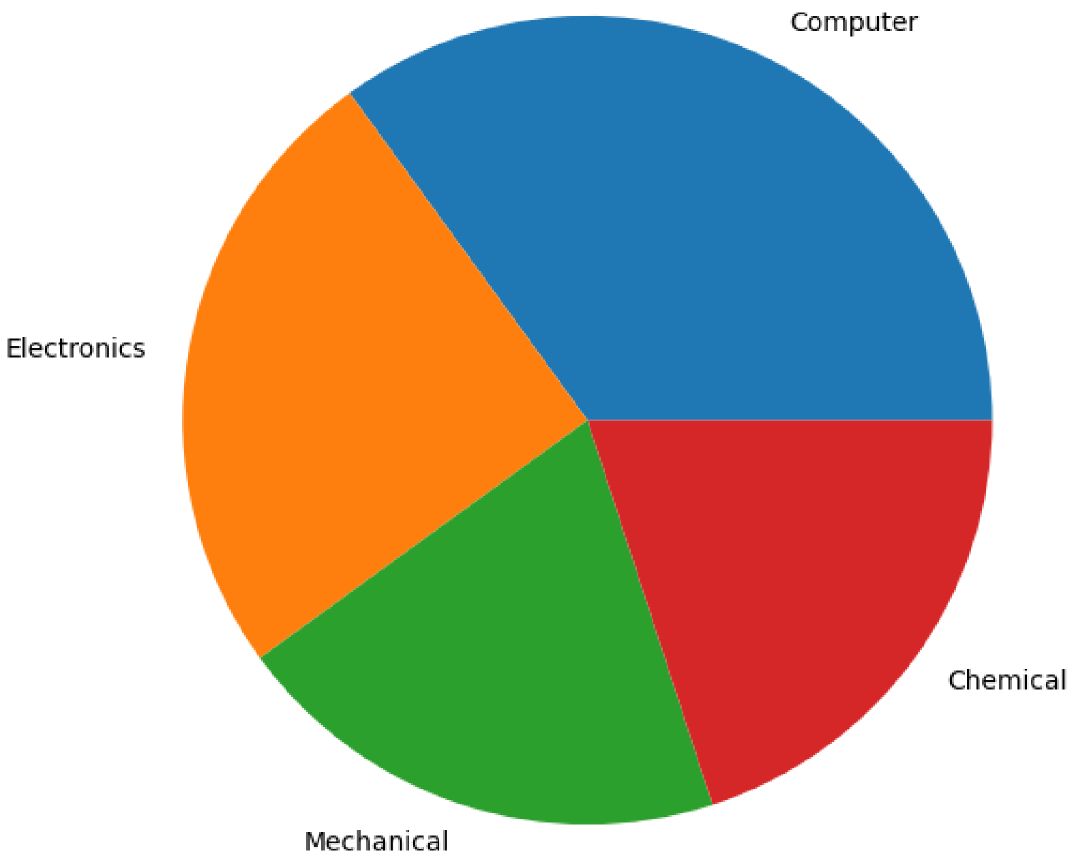
The optional bottom parameter of the plt.bar() function allows us to specify a starting position for a bar. Instead of running from zero to a value, it will go from the bottom to value. The first call to plt.bar() plots the blue bars. The second call to plt.bar() plots the red bars, with the bottom of the red bars being at the top of the blue bars.

## 23 Pie Chart

Pie charts are circular representations, divided into sectors. The sectors are also called wedges. The arc length of each sector is proportional to the quantity we are describing. It is an effective way to represent information when we are interested mainly in comparing the wedge against the whole pie, instead of wedges against each other.

Matplotlib provides the pie() function to plot pie charts from an array X. Wedges are created proportionally, so that each value x of array X generates a wedge proportional to x/sum(X).

[51]:
```
plt.figure(figsize=(7,7))

x10 = [35, 25, 20, 20]

labels = ['Computer', 'Electronics', 'Mechanical', 'Chemical']

plt.pie(x10, labels=labels);
plt.show()
```

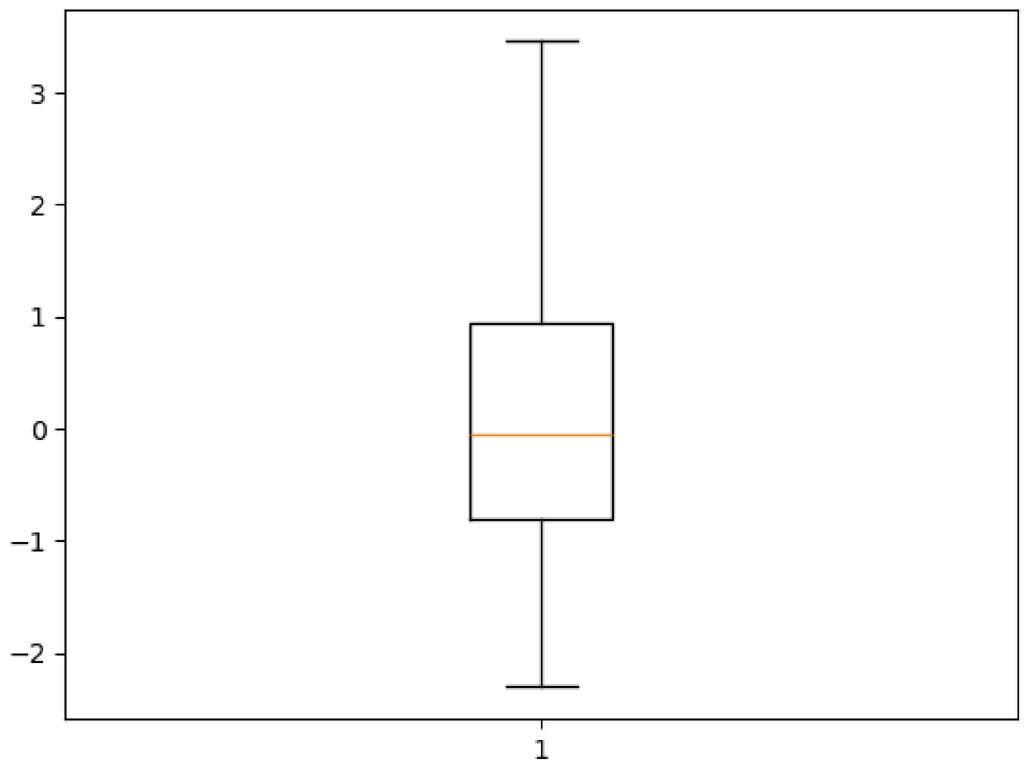


## 24 Boxplot

Boxplot allows us to compare distributions of values by showing the median, quartiles, maximum and minimum of a set of values.

We can plot a boxplot with the boxplot() function as follows:-

[52]:
```
data3 = np.random.randn(100)

plt.boxplot(data3)

plt.show();
```



The boxplot() function takes a set of values and computes the mean, median and other statistical quantities. The following points describe the preceeding boxplot:

- The red bar is the median of the distribution.
- The blue box includes 50 percent of the data from the lower quartile to the upper quartile. Thus, the box is centered on the median of the data.
- The lower whisker extends to the lowest value within 1.5 IQR from the lower quartile.
- The upper whisker extends to the highest value within 1.5 IQR from the upper quartile.
- Values further from the whiskers are shown with a cross marker.

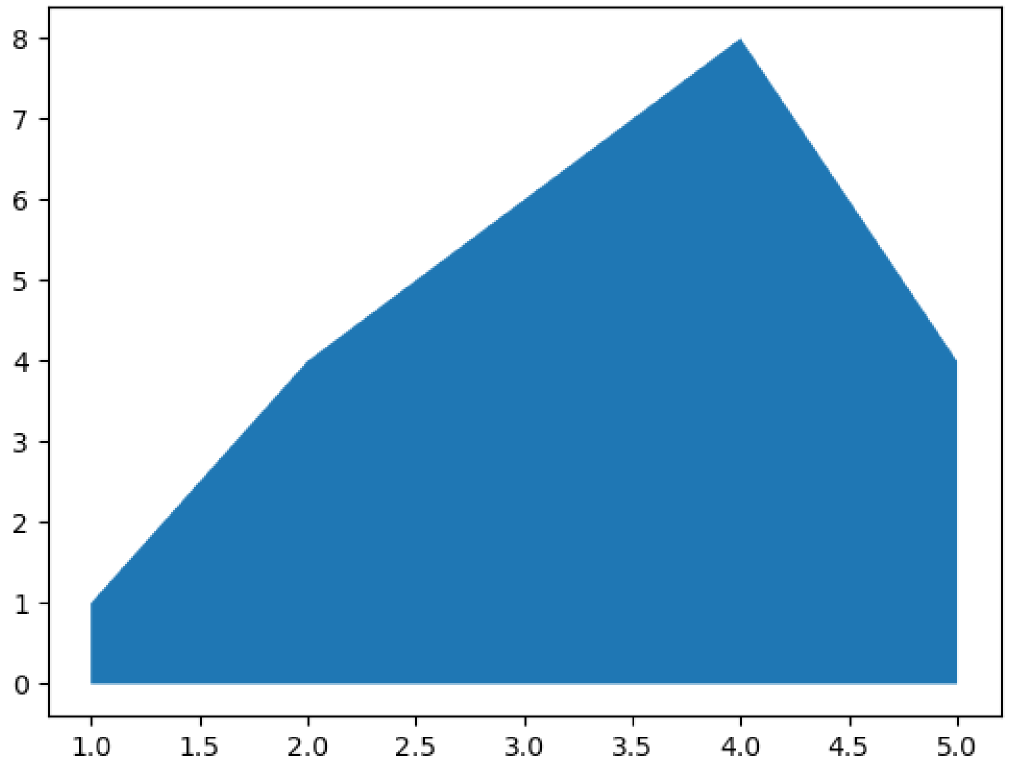
## 25 Area Chart

An Area Chart is very similar to a Line Chart. The area between the x-axis and the line is filled in with color or shading. It represents the evolution of a numerical variable following another numerical variable.

We can create an Area Chart as follows:-

[53]:
```
# Create some data
x12 = range(1, 6)
y12 = [1, 4, 6, 8, 4]
```

```python
# Area plot
plt.fill_between(x12, y12)
plt.show()
```



I have created a basic Area chart. I could also use the stackplot function to create the Area chart as follows:- `python plt.stackplot(x12, y12)` The fill_between() function is more convenient for future customization.

## 26 Contour Plot

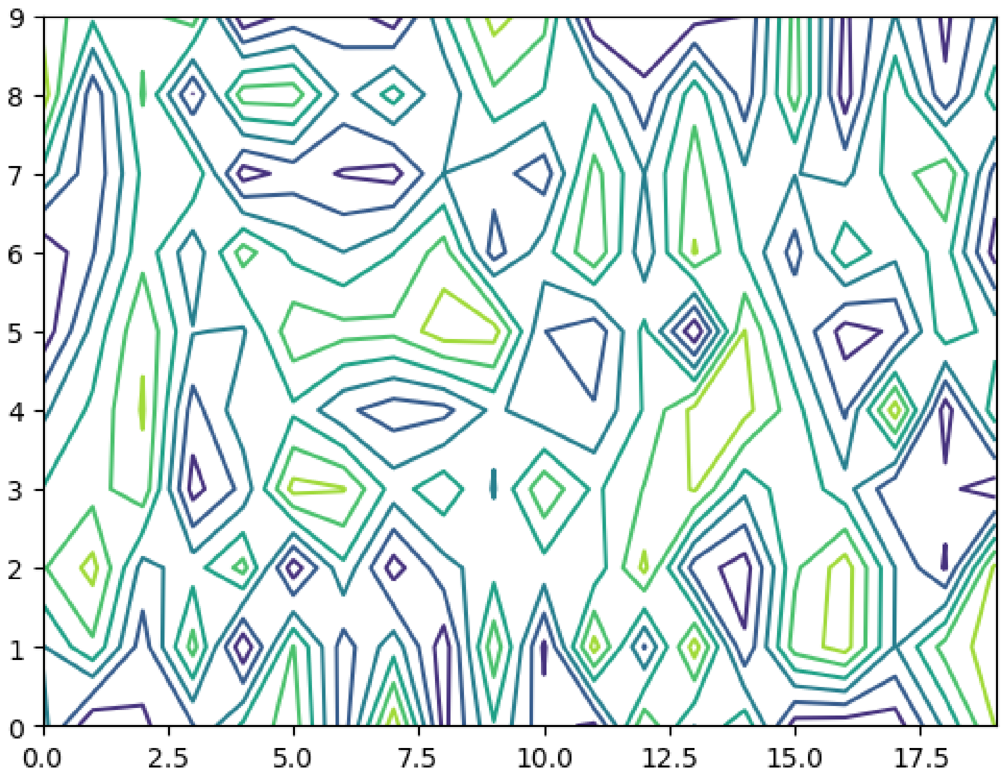
Contour plots are useful to display three-dimensional data in two dimensions using contours or color-coded regions. Contour lines are also known as level lines or isolines. Contour lines for a function of two variables are curves where the function has constant values. They have specific names beginning with iso- according to the nature of the variables being mapped.

There are lot of applications of Contour lines in several fields such as meteorology(for temperature, pressure, rain, wind speed), geography, magnetism, engineering, social sciences and so on.

The density of the lines indicates the slope of the function. The gradient of the function is always perpendicular to the contour lines. When the lines are close together, the length of the gradient is large and the variation is steep.

A Contour plot can be created with the plt.contour() function as follows:-

[54]:
```
# Create a matrix
matrix1 = np.random.rand(10, 20)

cp = plt.contour(matrix1)

plt.show()
```



The contour() function draws contour lines. It takes a 2D array as input.Here, it is a matrix of 10 x 20 random elements.

The number of level lines to draw is chosen automatically, but we can also specify it as an additional parameter, N.

`plt.contour(matrix, N)`

## 27 Styles with Matplotlib Plots

The Matplotlib version 1.4 which was released in August 2014 added a very convenient style module. It includes a number of new default stylesheets, as well as the ability to create and package own styles.

We can view the list of all available styles by the following command. "' python print(plt.style.availabe)

[56]:
```python
# View list of all available styles

print(plt.style.available)
```

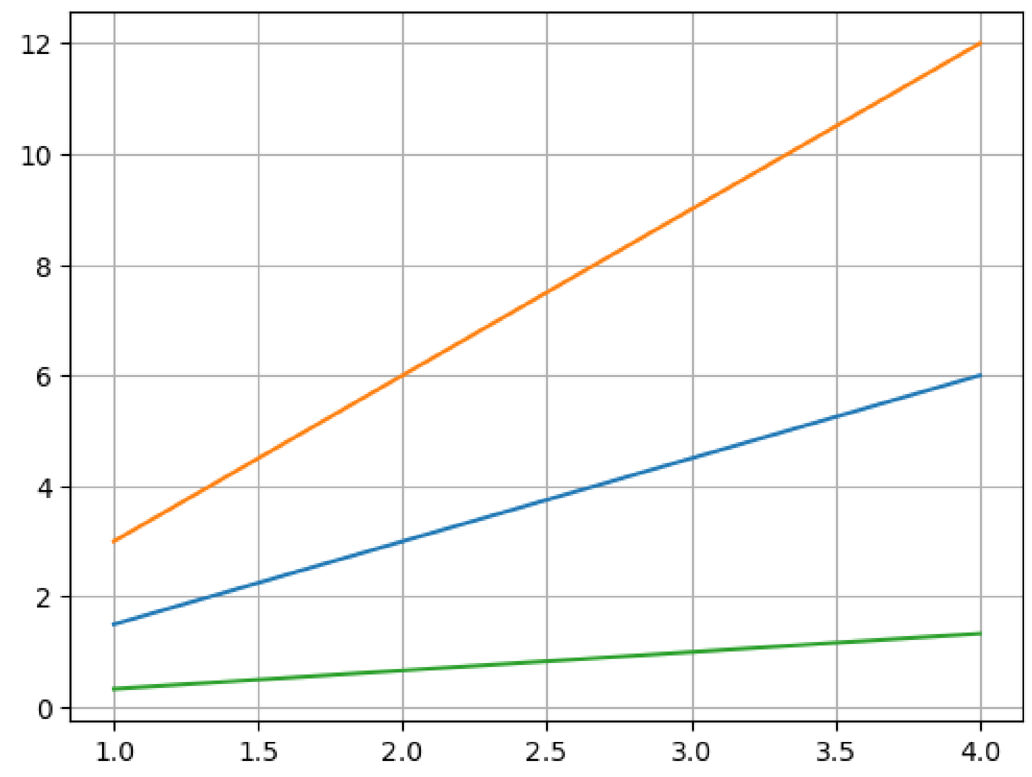
```
['Solarize_Light2', '_classic_test_patch', '_mpl-gallery', '_mpl-gallery-
nogrid', 'bmh', 'classic', 'dark_background', 'fast', 'fivethirtyeight',
'ggplot', 'grayscale', 'seaborn-v0_8', 'seaborn-v0_8-bright',
'seaborn-v0_8-colorblind', 'seaborn-v0_8-dark', 'seaborn-v0_8-dark-palette',
'seaborn-v0_8-darkgrid', 'seaborn-v0_8-deep', 'seaborn-v0_8-muted',
'seaborn-v0_8-notebook', 'seaborn-v0_8-paper', 'seaborn-v0_8-pastel',
'seaborn-v0_8-poster', 'seaborn-v0_8-talk', 'seaborn-v0_8-ticks',
'seaborn-v0_8-white', 'seaborn-v0_8-whitegrid', 'tableau-colorblind10']
```

We can set the Styles for Matplotlib plots as follows:- "'python plt.style.use('seaborn-bright')

## 27.1 Adding a grid

In some cases, the background of a plot was completely blank. We can get more information, if there is a reference system in the plot. The reference system would improve the comprehension of the plot. An example of the reference system is adding a grid. We can add a grid to the plot by calling the grid() function. It takes one parameter, a Boolean value, to enable(if True) or disable(if False) the grid.

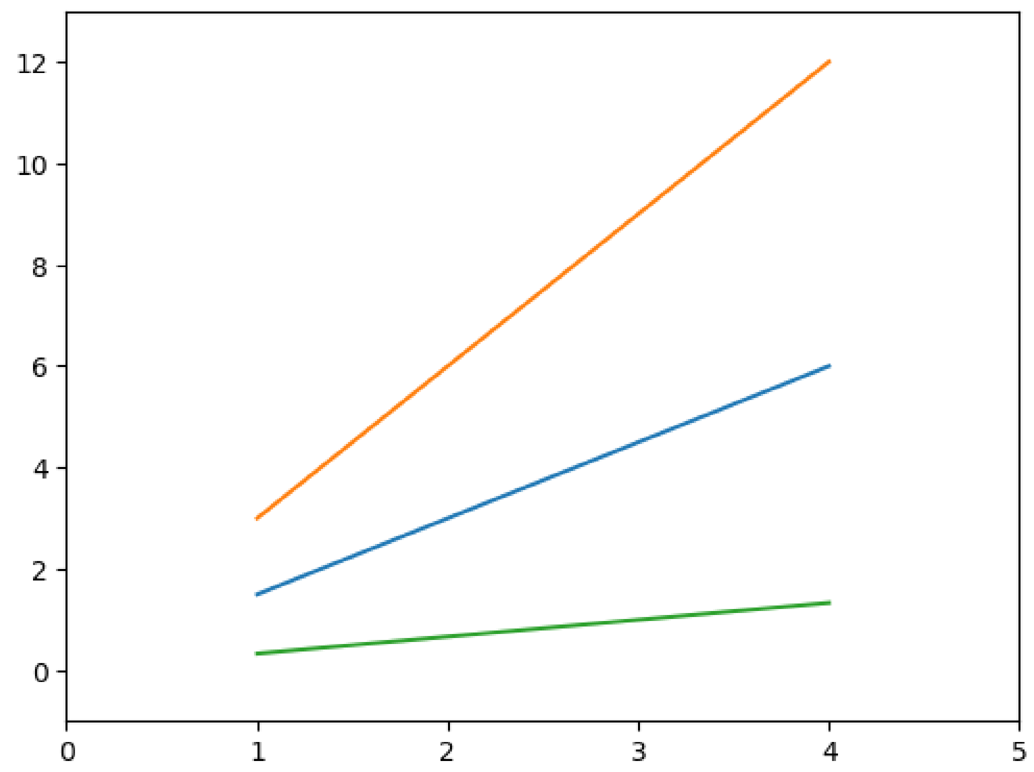
[58]:
```python
x15 = np.arange(1, 5)

plt.plot(x15, x15*1.5, x15, x15*3.0, x15, x15/3.0)

plt.grid(True)

plt.show()
```



## 27.2 Handling axes

Matplotlib automatically sets the limits of the plot to precisely contain the plotted datasets. Sometimes, we want to set the axes limits ourself. We can set the axes limits with the axis() function as follows:-

[59]:
```
x15 = np.arange(1, 5)

plt.plot(x15, x15*1.5, x15, x15*3.0, x15, x15/3.0)

plt.axis()   # shows the current axis limits values

plt.axis([0, 5, -1, 13])

plt.show()
```



We can see that we now have more space around the lines.

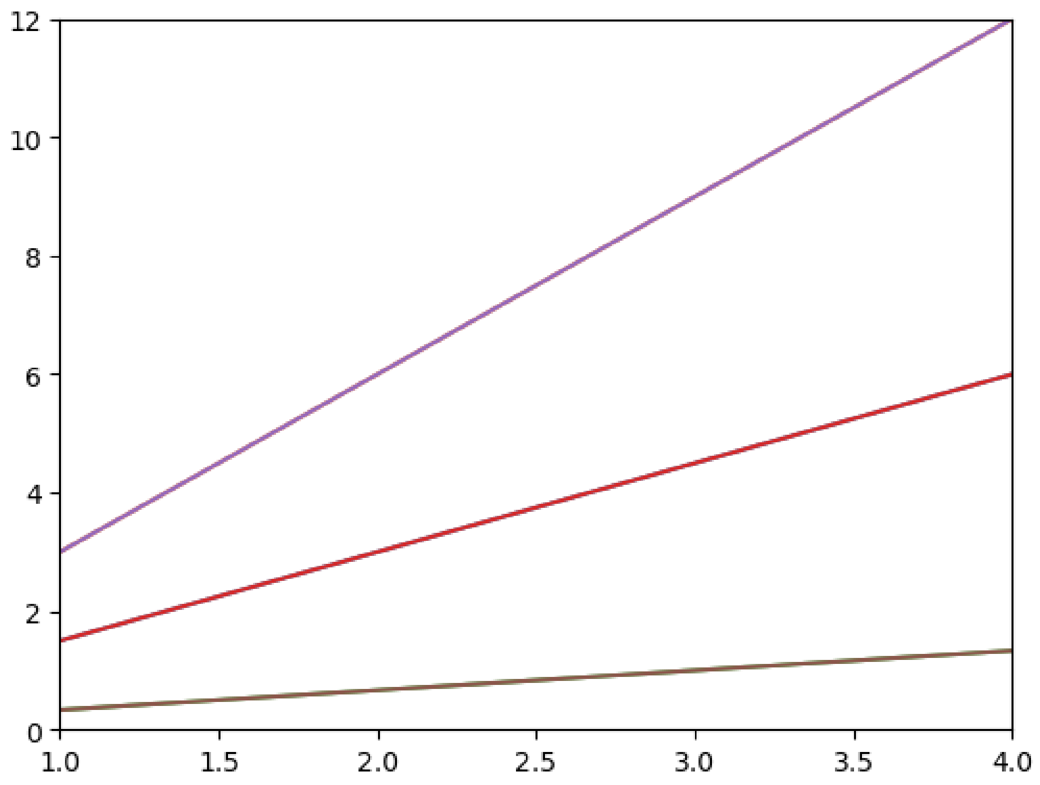
If we execute axis() without parameters, it returns the actual axis limits.

We can set parameters to axis() by a list of four values.

The list of four values are the keyword arguments [xmin, xmax, ymin, ymax] allows the minimum and maximum limits for X and Y axis respectively.

We can control the limits for each axis separately using the xlim() and ylim() functions. This can be done as follows:-

[61]:
```
x15 = np.arange(1, 5)

plt.plot(x15, x15*1.5, x15, x15*3.0, x15, x15/3.0)

plt.xlim([1.0, 4.0])

plt.ylim([0.0, 12.0])
plt.show()
```



## 27.3 Handling X and Y ticks

Vertical and horizontal ticks are those little segments on the axes, coupled with axes labels, used to give a reference system on the graph.So, they form the origin and the grid lines.

Matplotlib provides two basic functions to manage them - xticks() and yticks().

Executing with no arguments, the tick function returns the current ticks' locations and the labels corresponding to each of them.

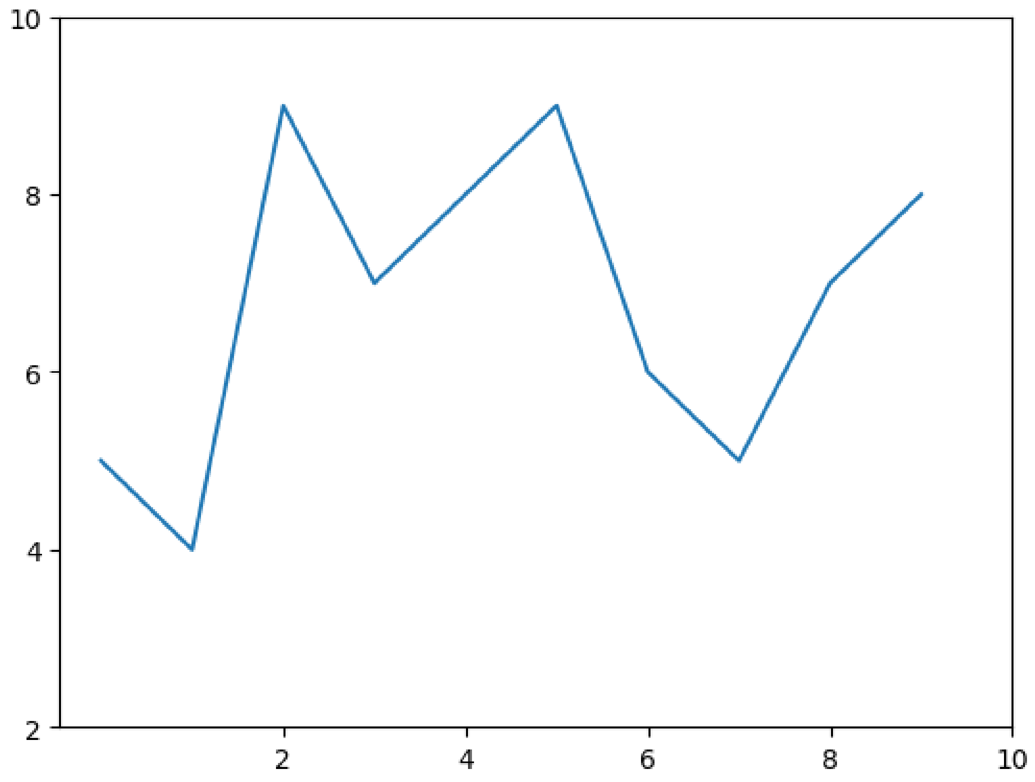
We can pass arguments(in the form of lists) to the ticks functions. The arguments are:-

Locations of the ticks

Labels to draw at these locations.

We can demonstrate the usage of the ticks functions in the code snippet below:-

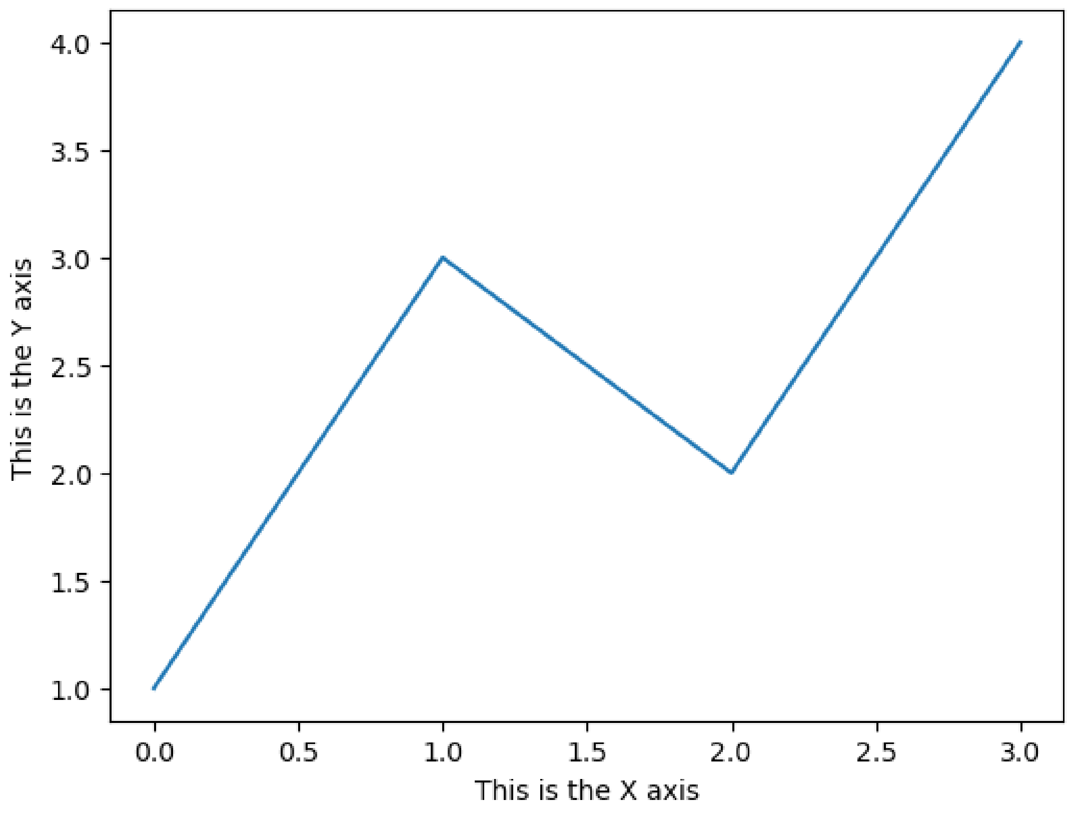
[62]:
```
u = [5, 4, 9, 7, 8, 9, 6, 5, 7, 8]

plt.plot(u)

plt.xticks([2, 4, 6, 8, 10])
plt.yticks([2, 4, 6, 8, 10])
plt.show()
```



## 27.4 Adding labels

Another important piece of information to add to a plot is the axes labels, since they specify the type of data we are plotting.

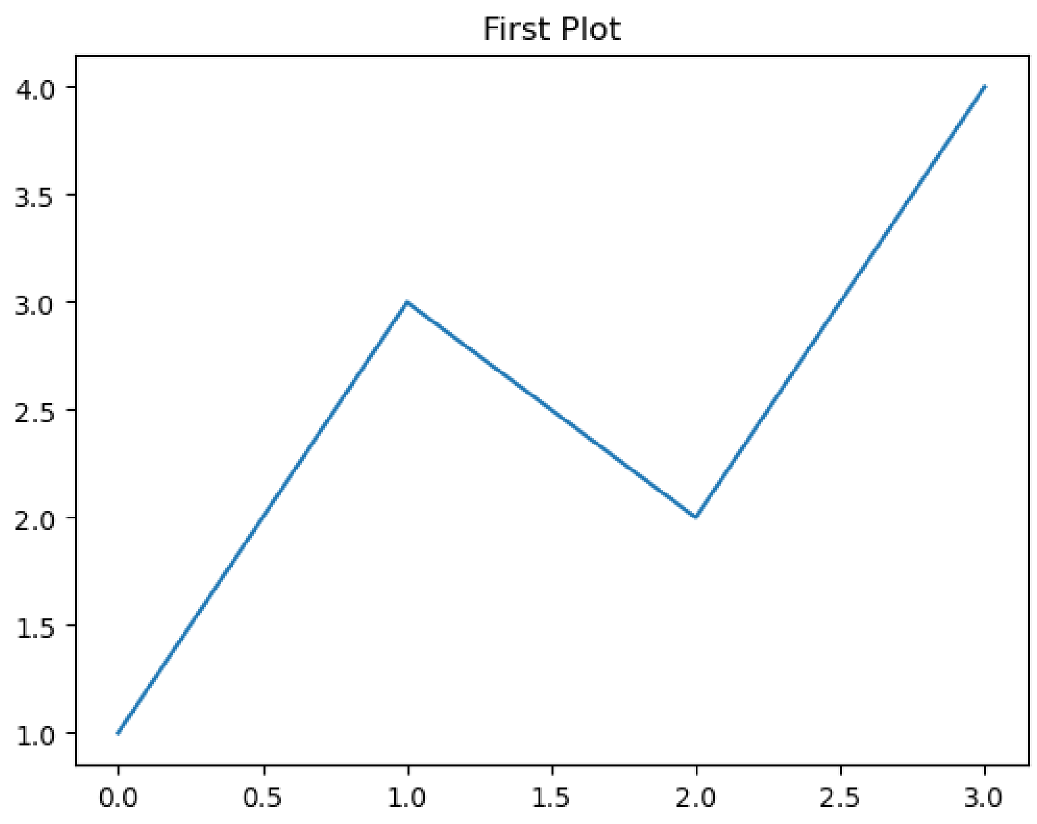
[63]:
```
plt.plot([1, 3, 2, 4])

plt.xlabel('This is the X axis')

plt.ylabel('This is the Y axis')

plt.show()
```



### 27.5 Adding a title

The title of a plot describes about the plot. Matplotlib provides a simple function title() to add a title to an image.

```
[64]: plt.plot([1, 3, 2, 4])

      plt.title('First Plot')

      plt.show()
```



The above plot displays the output of the previous code. The title First Plot is displayed on top of the plot.

## 27.6 Adding a legend

Legends are used to describe what each line or curve means in the plot.

Legends for curves in a figure can be added in two ways. One method is to use the legend method of the axis object and pass a list/tuple of legend texts as follows:-

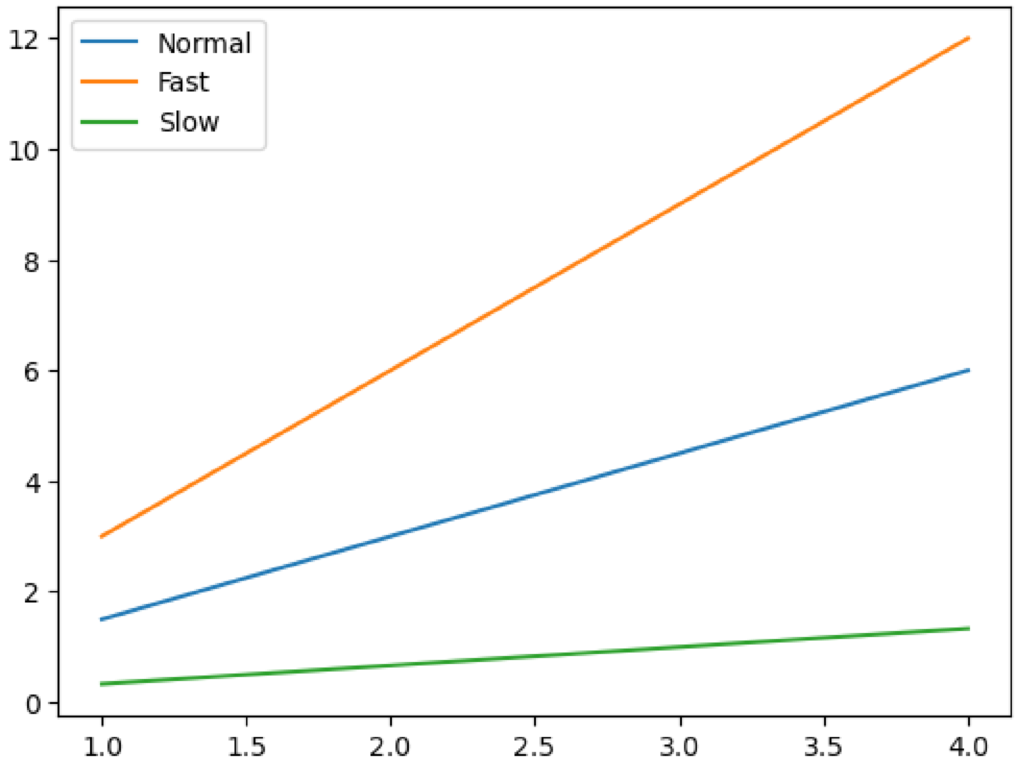
[65]:
```
x15 = np.arange(1, 5)

fig, ax = plt.subplots()

ax.plot(x15, x15*1.5)
ax.plot(x15, x15*3.0)
ax.plot(x15, x15/3.0)

ax.legend(['Normal','Fast','Slow']);
```

The above method follows the MATLAB API. It is prone to errors and unflexible if curves are added to or removed from the plot. It resulted in a wrongly labelled curve.

A better method is to use the label keyword argument when plots are added to the figure. Then we use the legend method without arguments to add the legend to the figure.

The advantage of this method is that if curves are added or removed from the figure, the legend is automatically updated accordingly. It can be achieved by executing the code below:-

[68]:
```
x15 = np.arange(1, 5)

fig, ax = plt.subplots()

ax.plot(x15, x15*1.5, label='Normal')
ax.plot(x15, x15*3.0, label='Fast')
ax.plot(x15, x15/3.0, label='Slow')

ax.legend();
plt.show()
```



The legend function takes an optional keyword argument loc. It specifies the location of the legend to be drawn. The loc takes numerical codes for the various places the legend can be drawn. The most common loc values are as follows:-

ax.legend(loc=0) # let Matplotlib decide the optimal location

ax.legend(loc=1) # upper right corner

ax.legend(loc=2) # upper left corner

ax.legend(loc=3) # lower left corner

ax.legend(loc=4) # lower right corner

ax.legend(loc=5) # right

ax.legend(loc=6) # center left

ax.legend(loc=7) # center right

ax.legend(loc=8) # lower center

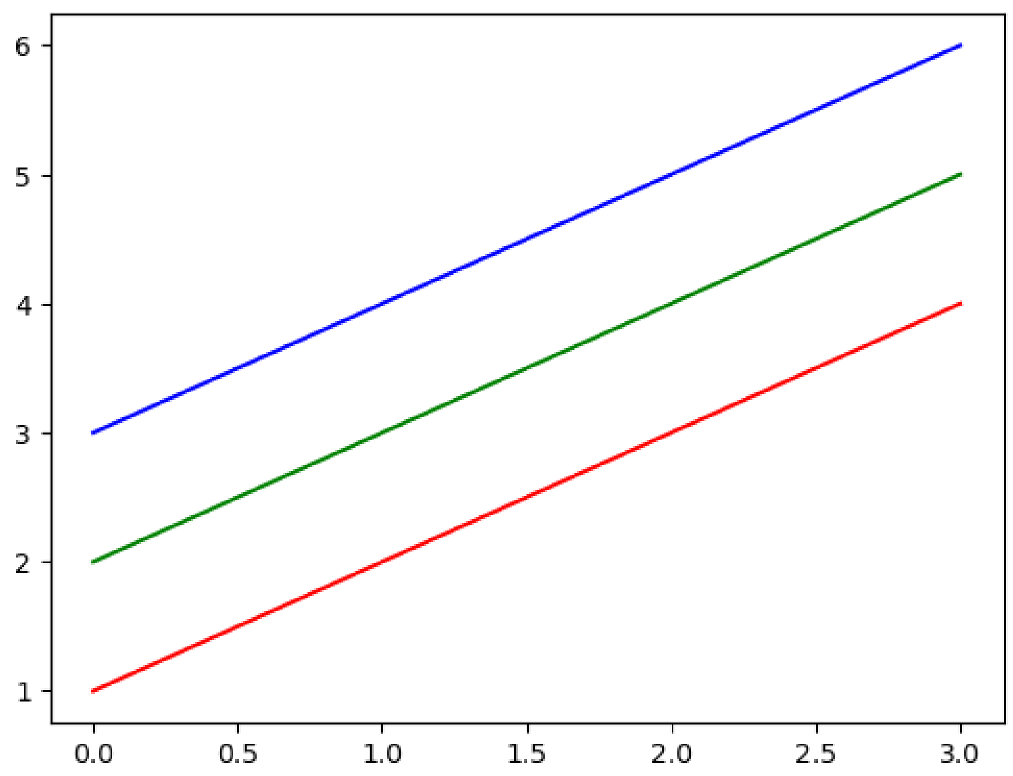
ax.legend(loc=9) # upper center

ax.legend(loc=10) # center

## 27.7 Control colours

We can draw different lines or curves in a plot with different colours. In the code below, we specify colour as the last argument to draw red, blue and green lines.

[69]:
```
x16 = np.arange(1, 5)

plt.plot(x16, 'r')
plt.plot(x16+1, 'g')
plt.plot(x16+2, 'b')

plt.show()
```



The colour names and colour abbreviations are given in the following table:-

| Colour abbreviation | Colour name |
|---|---|
| b | blue |
| c | cyan |
| g | green |
| k | black |
| m | magenta |
| r | red |
| w | white |
| y | yellow |

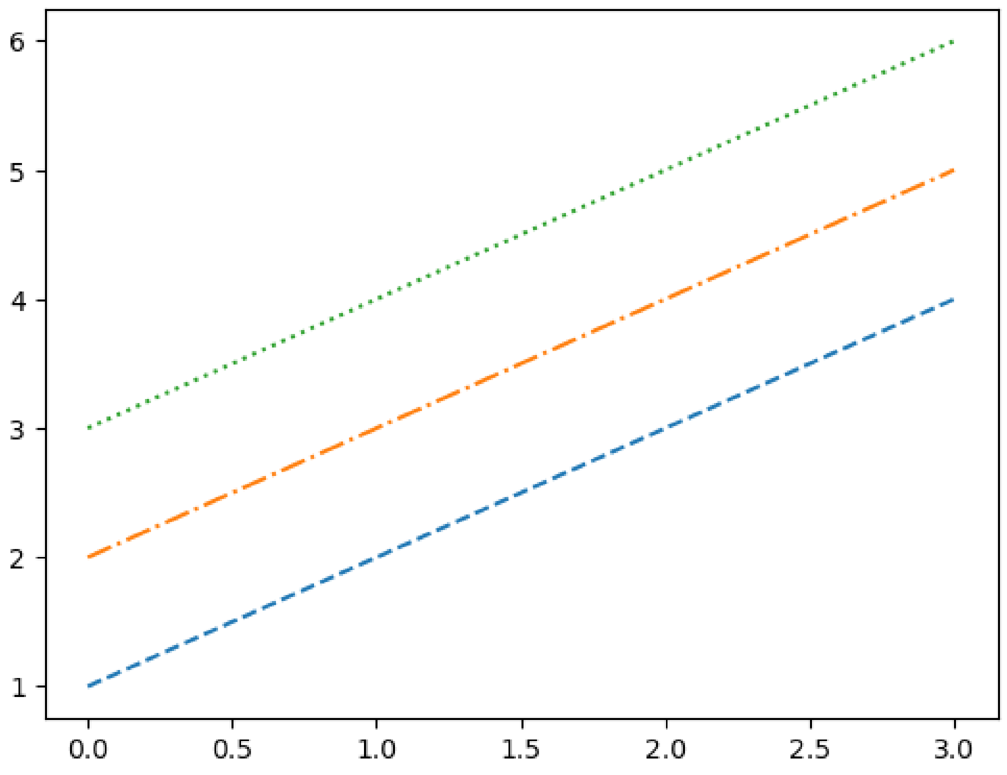
There are several ways to specify colours, other than by colour abbreviations:

- The full colour name, such as yellow
- Hexadecimal string such as ##FF00FF
- RGB tuples, for example (1, 0, 1)
- Grayscale intensity, in string format such as '0.7'.

### 27.8 Control line styles

Matplotlib provides us different line style options to draw curves or plots. In the code below, I use different line styles to draw different plots.

[70]:
```
x16 = np.arange(1, 5)

plt.plot(x16, '--', x16+1, '-.', x16+2, ':')

plt.show()
```



The above code snippet generates a blue dashed line, a green dash-dotted line and a red dotted line.

All the available line styles are available in the following table:

Style abbreviation Style

solid line

– dashed line

**-. dash-dot line** dotted line

Now, we can see the default format string for a single line plot is 'b-'.

## 28 Summary

In this project, I discuss Matplotlib (the basic plotting library in Python) and throw some light on various charts and customization techniques associated with it.

In particular, I discuss Matplotlib object hierarchy, Matplotlib architecture, Pyplot and Object-Oriented architecture. I also discuss subplots which is very important tool to create graphics in Matplotlib.

Then, I discuss various types of plots like line plot, scatter plot, histogram, bar chart, pie chart, box plot, area chart and contour plot.

Finally, I discuss various customization techniques. I discuss how to customize the graphics with styles. I discuss how to add a grid and how to handle axes and ticks. I discuss how to add labels, title and legend. I discuss how to customize the charts with colours and line styles.