Introduction to Matplotlib

Matplotlib is the basic plotting library of Python programming language. It is the most prominent tool among Python visualization packages. Matplotlib is highly efficient in performing wide range of tasks. It can produce publication quality figures in a variety of formats. It can export visualizations to all of the common formats like PDF, SVG, JPG, PNG, BMP and GIF. It can create popular visualization types – line plot, scatter plot, histogram, bar chart, error charts, pie chart, box plot, and many more types of plot. Matplotlib also supports 3D plotting. Many Python libraries are built on top of Matplotlib. For example, pandas and Seaborn are built on Matplotlib. They allow to access Matplotlib's methods with less code.

The project Matplotlib was started by John Hunter in 2002. Matplotlib was originally started to visualize Electrocorticography (ECoG) data of epilepsy patients during post-doctoral research in Neurobiology. The open-source tool Matplotlib emerged as the most widely used plotting library for the Python programming language. It was used for data visualization during landing of the Phoenix spacecraft in 2008.

Different types of chart

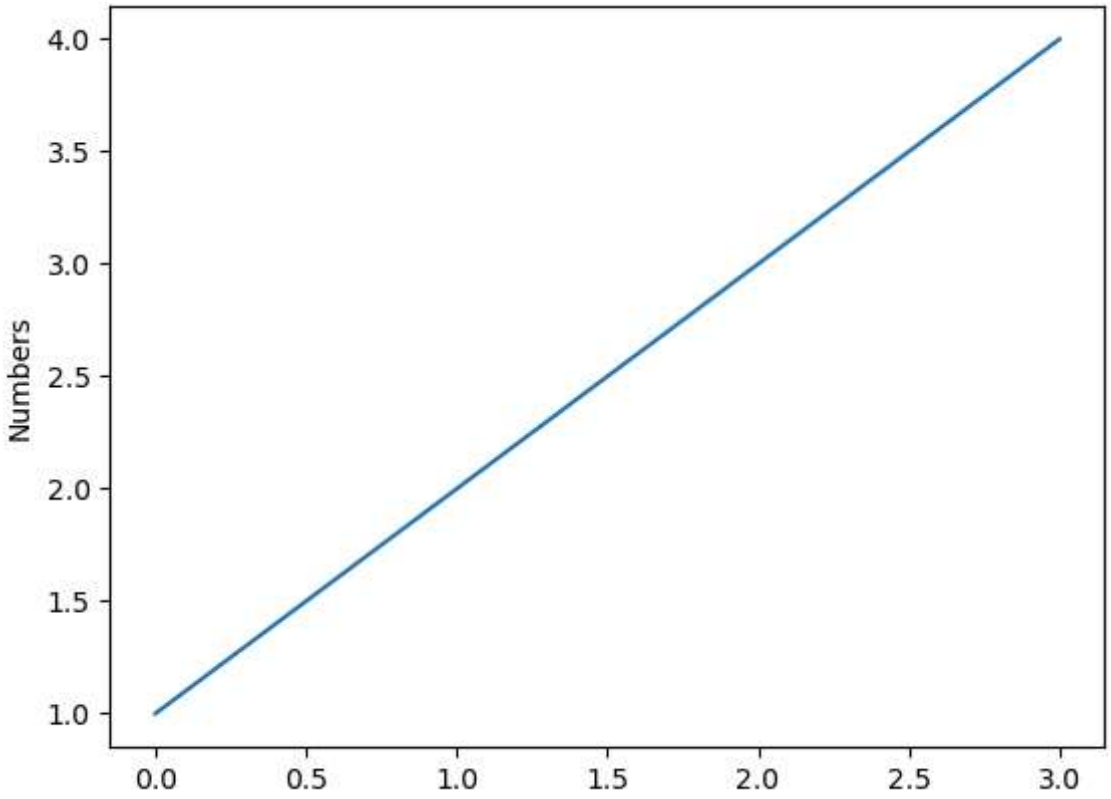
In [1]:
```
import pandas as pd
import numpy as np
```

In [ ]:
```
import matplotlib.pyplot as plt  # here we import the library
```

Visualization with Pyplot

Generating visualization with Pyplot is very easy. The x-axis values ranges from 0-3 and the y-axis from 1-4. If we provide a single list or array to the plot() command, matplotlib assumes it is a sequence of y values, and automatically generates the x values. Since python ranges start with 0, the default x vector has the same length as y but starts with 0. Hence the x data are [0,1,2,3] and y data are [1,2,3,4].

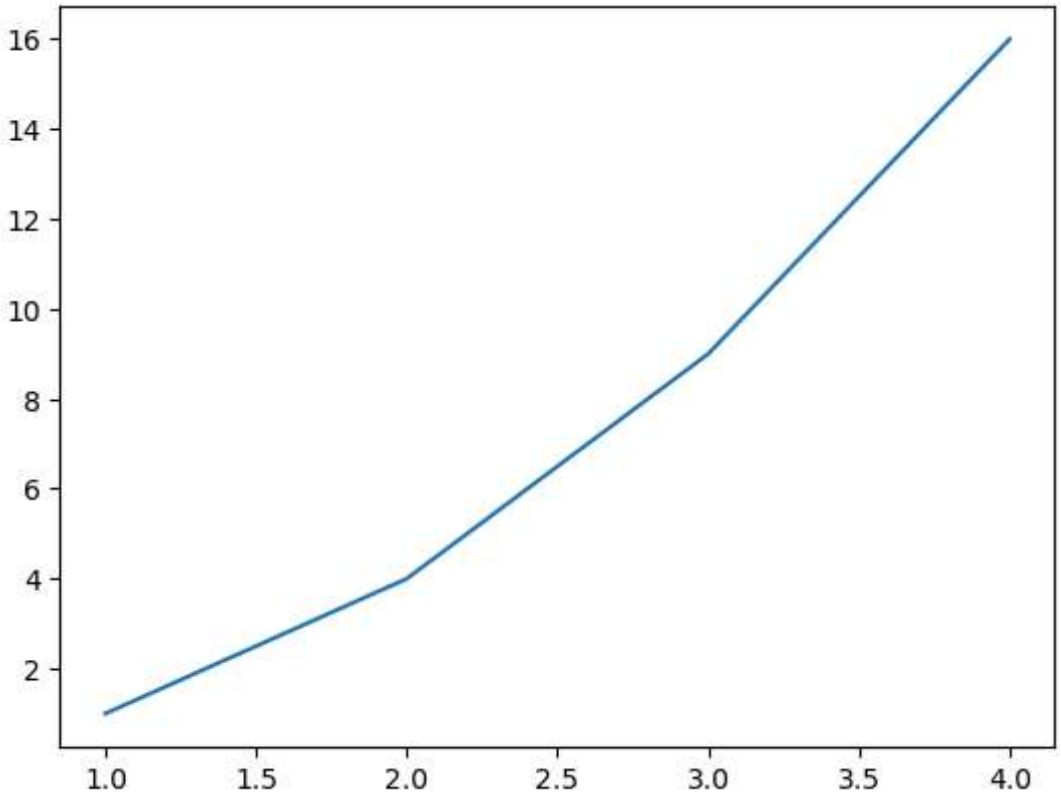
In [2]:
```
plt.plot([1, 2, 3, 4])
plt.ylabel('Numbers')
plt.show()
```



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:-

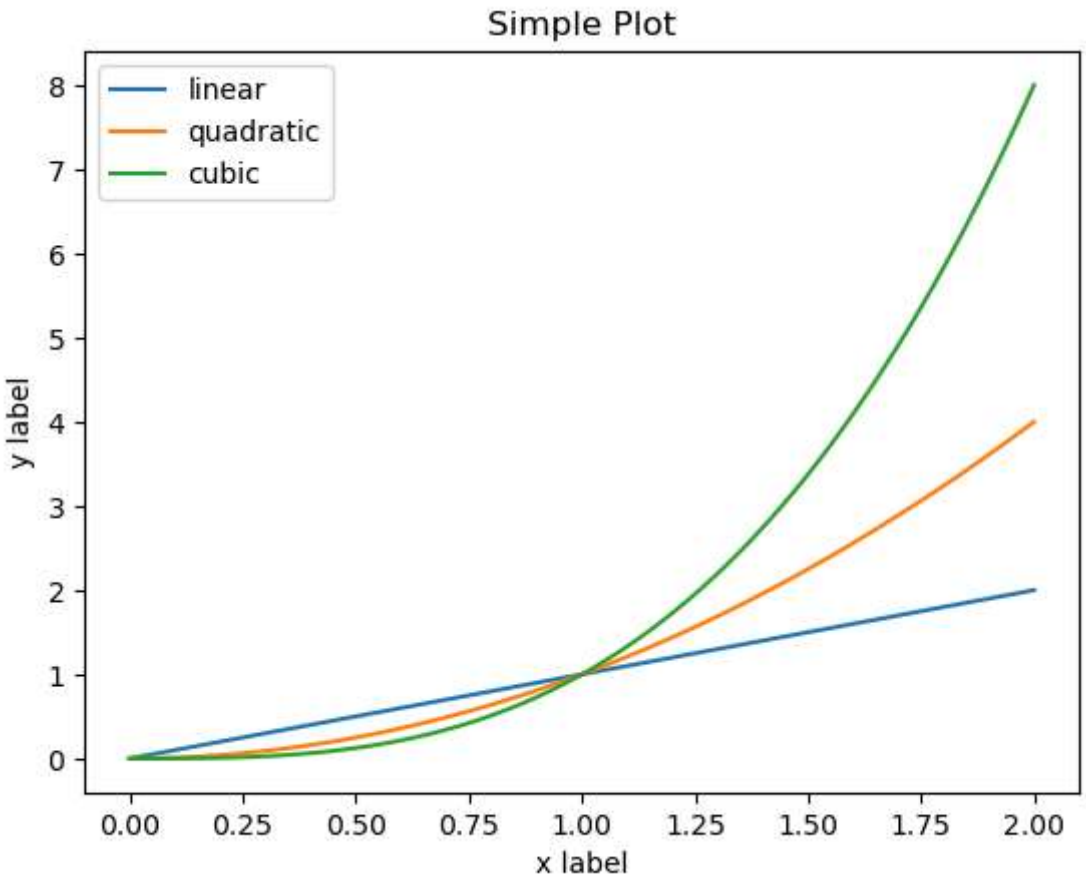
In [4]:
```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
plt.show()
```



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:

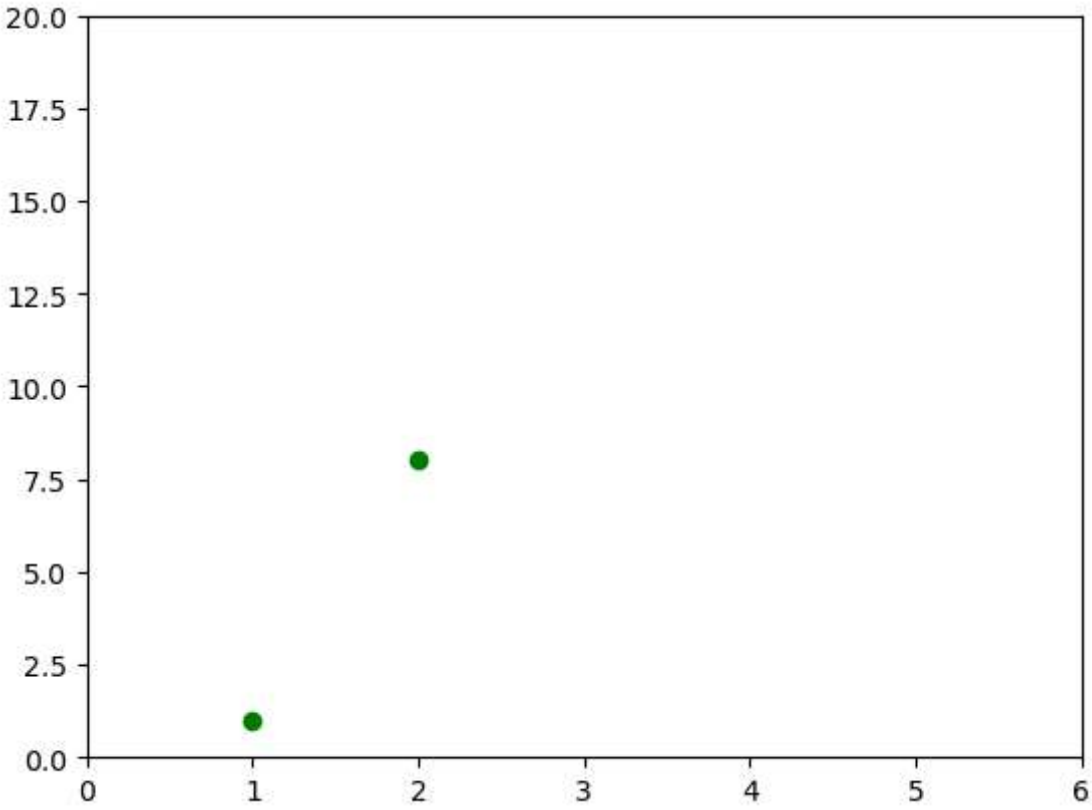
In [5]:
```
x = np.linspace(0, 2, 100)

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 plot

For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB. We can concatenate a color string with a line style string. The default format string is 'b-', which is a solid blue line. For example, to plot the above line with red circles, we would issue the following command

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



In [9]:
```
# 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()
```

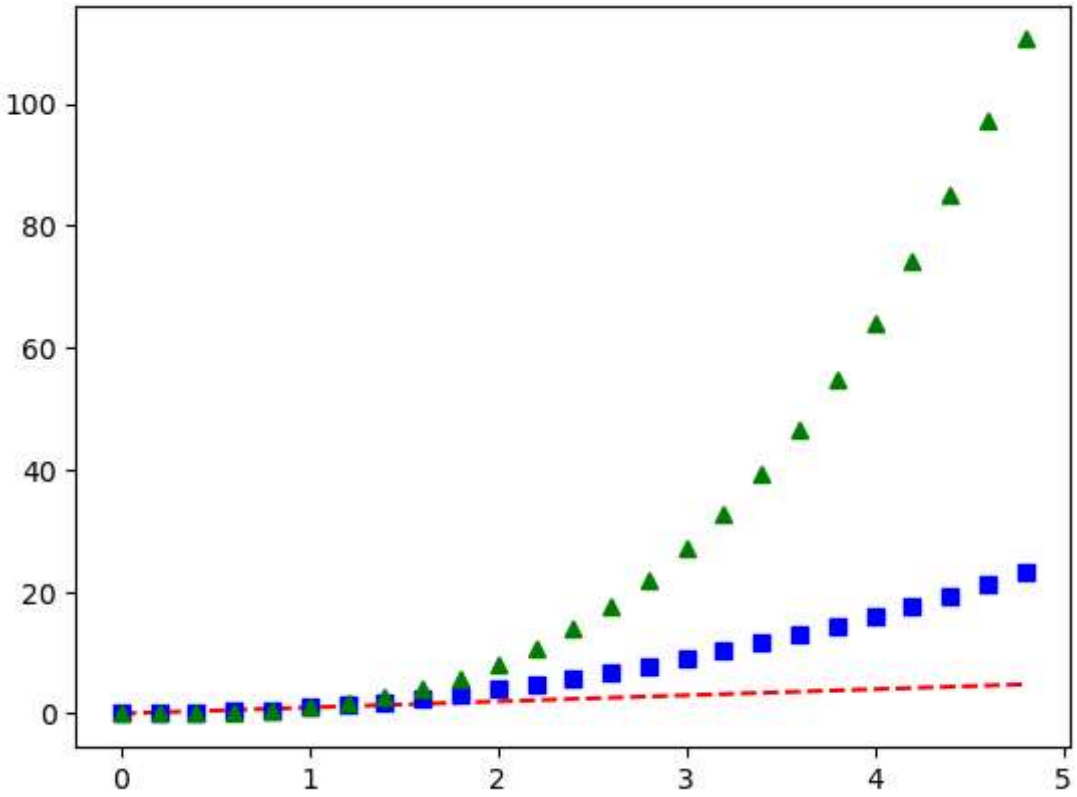


Figure and Axes

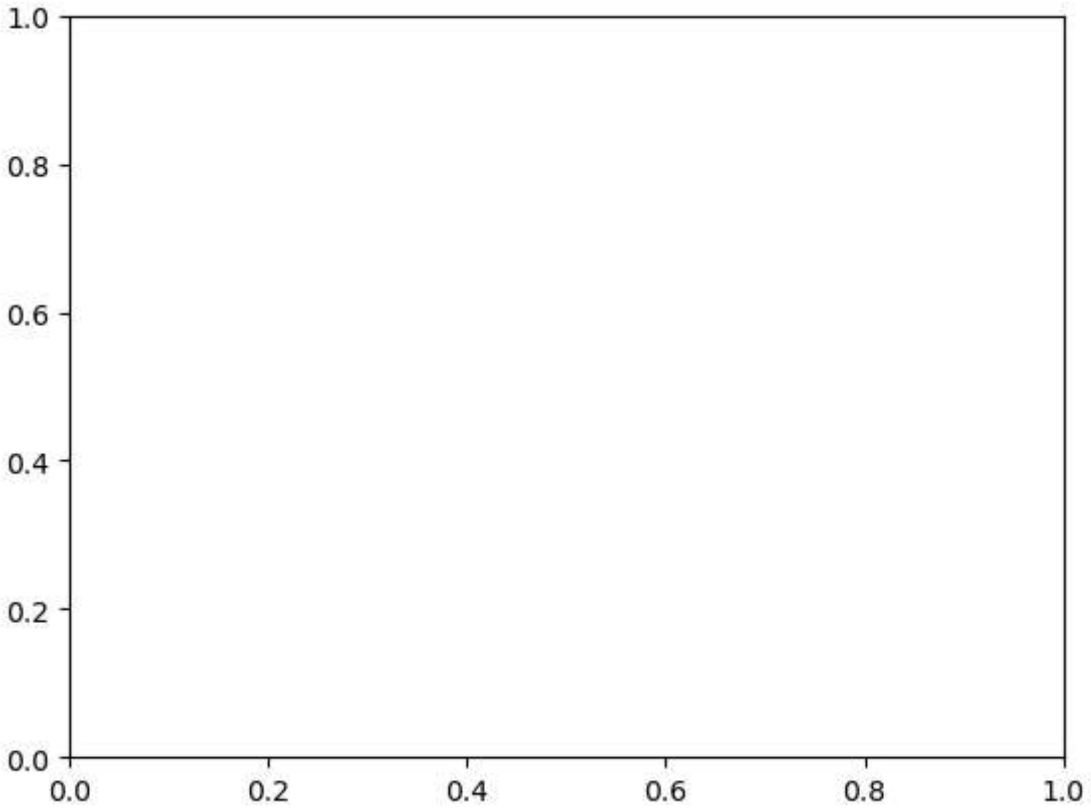
I start by creating a figure and an axes. A figure and axes can be created as follows:

fig = plt.figure()

ax = plt.axes()

In Matplotlib, the figure (an instance of the class plt.Figure) is a single container that contains all the objects representing axes, graphics, text and labels. The axes (an instance of the class plt.Axes) is a bounding box with ticks and labels. It will contain the plot elements that make up the visualization. I have used the variable name fig to refer to a figure instance, and ax to refer to an axes instance or group of axes instances.

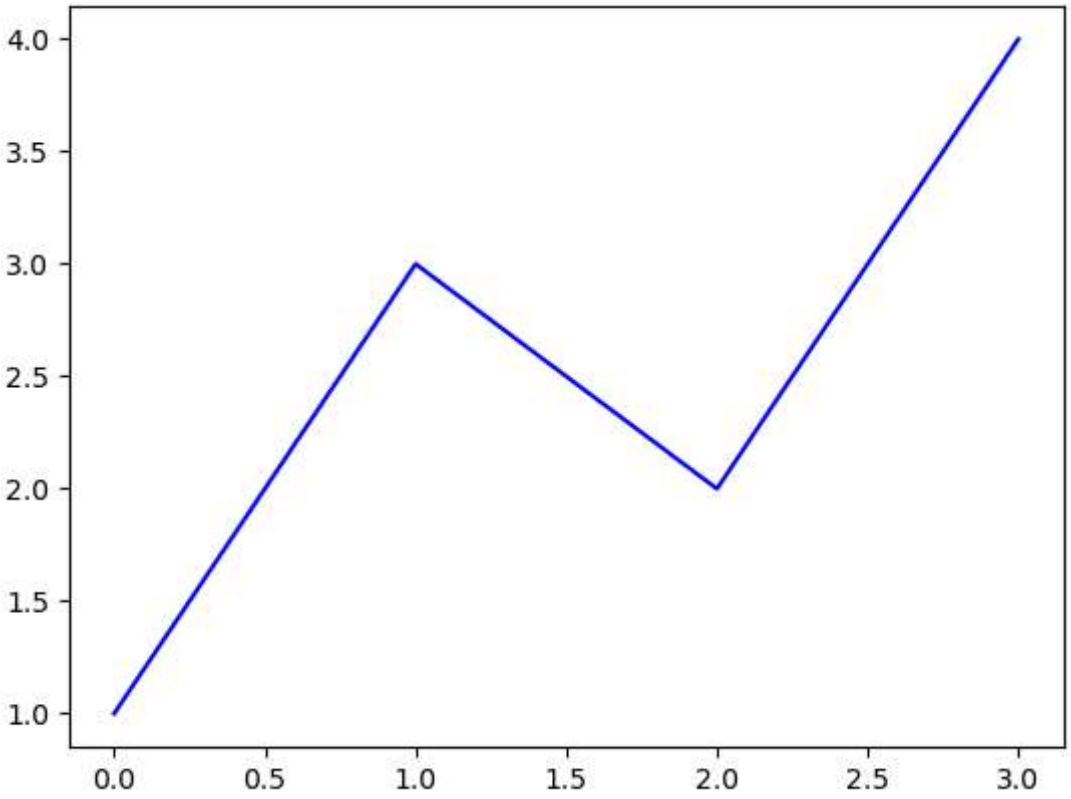
In [14]:
```
fig = plt.figure()

ax = plt.axes()
```



First plot with Matplotlib

Now, I will start producing plots.

In [15]:
```
plt.plot([1, 3, 2, 4], 'b-')

plt.show( )
```



Specify both Lists

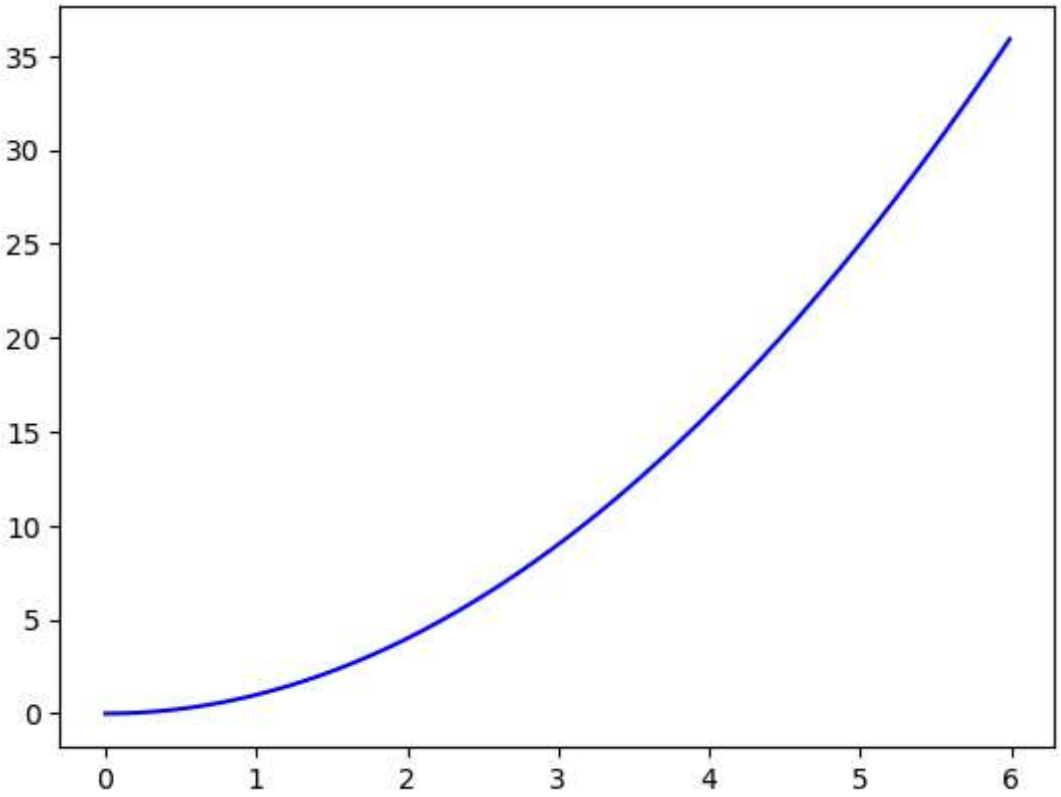
Also, we can explicitly specify both the lists as follows:-

x3 = range(6)

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

plt.show()

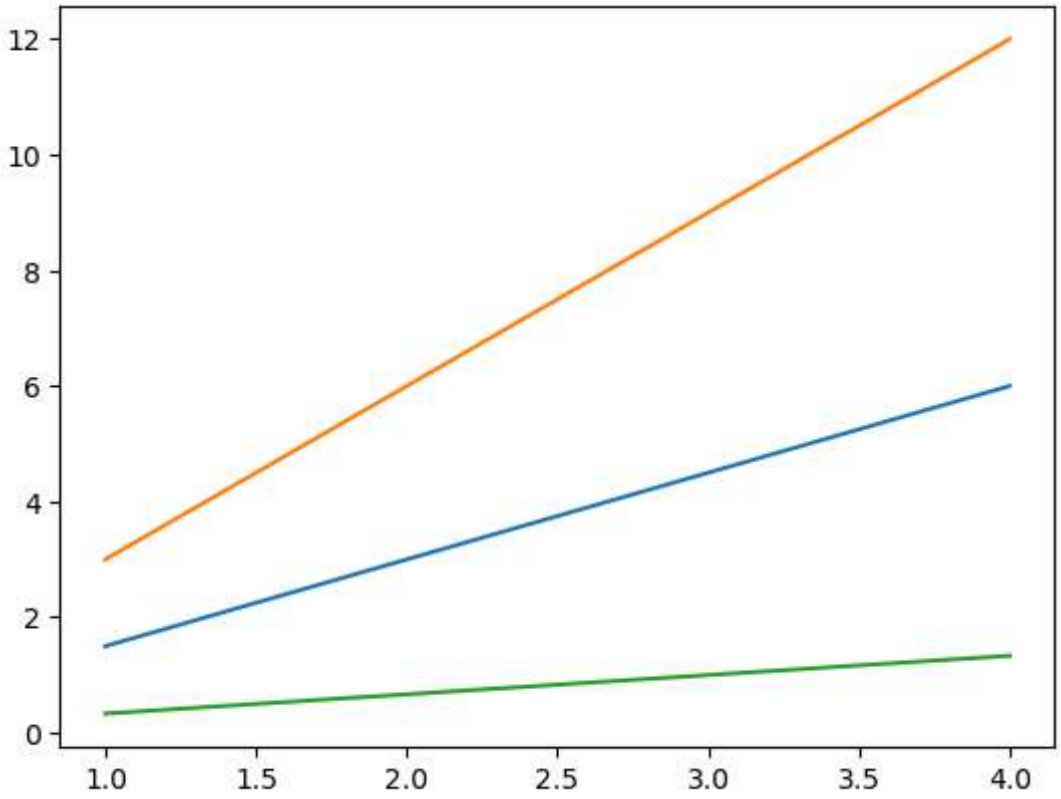
In [16]:
```
x3 = np.arange(0.0, 6.0, 0.01)

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

plt.show()
```



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:-

In [17]:
```
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()
```



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')

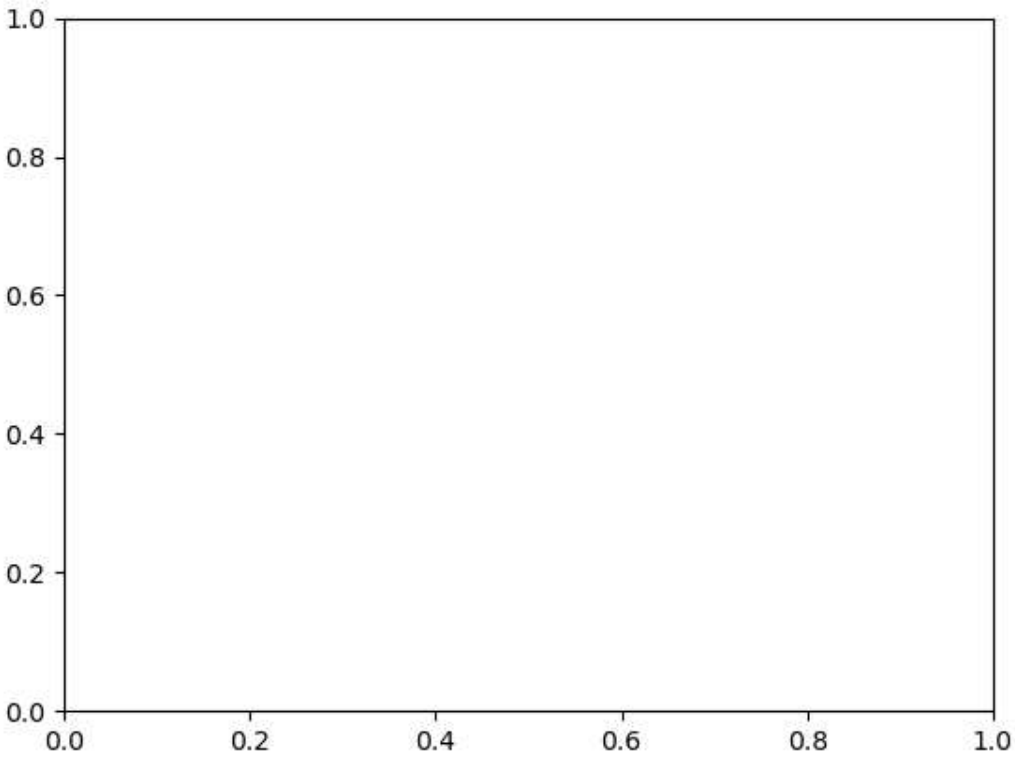
In savefig() command, the file format is inferred from the extension of the given filename. Depending on the backend, many different file formats are available. The list of supported file types can be found by using the get_supported_filetypes() method of the figure canvas object as follows:-

fig.canvas.get_supported_filetypes()

In [18]:
```
# Saving the figure

fig.savefig('plot1.png')

# Explore the contents of figure
```

```
from IPython.display import Image

Image('plot1.png')
```

Out[18]:



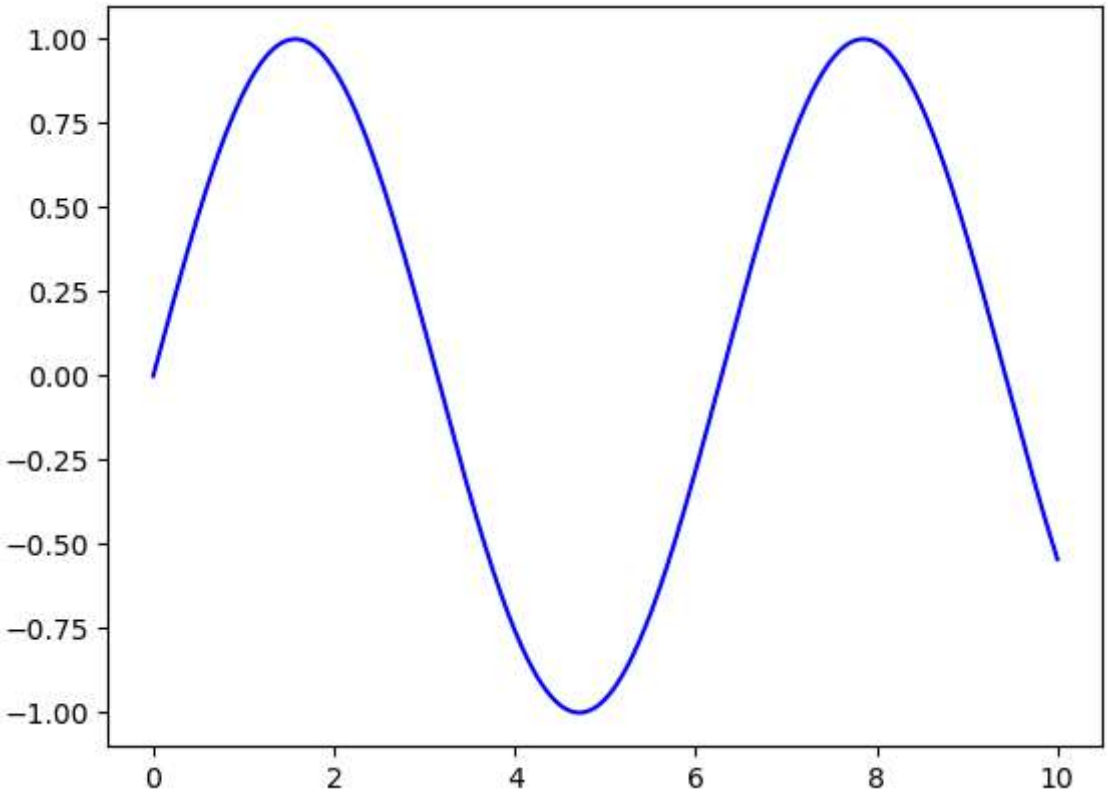
In [19]:
```
# Explore supported file formats
fig.canvas.get_supported_filetypes()
```

Out[19]:
```
{'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'}
```

Line Plot

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

In [20]:
```
# 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-');
```

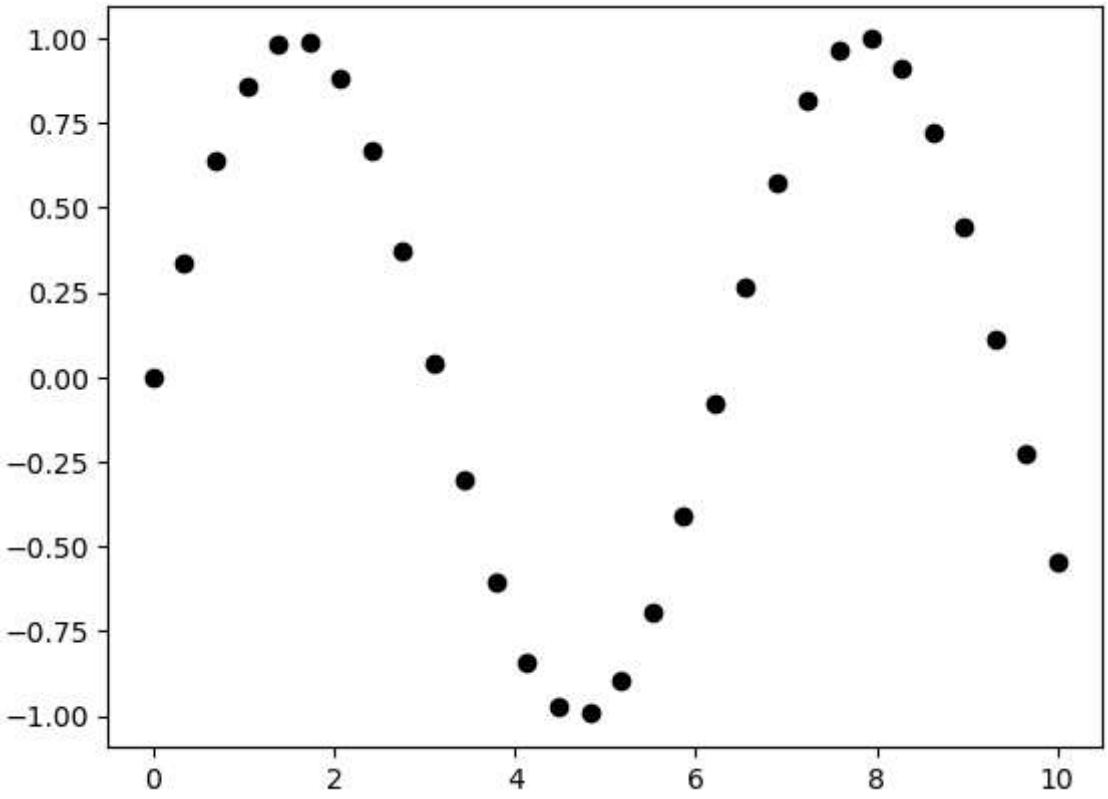


Scatter Plot

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:

In [21]:
```
x7 = np.linspace(0, 10, 30)

y7 = np.sin(x7)

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

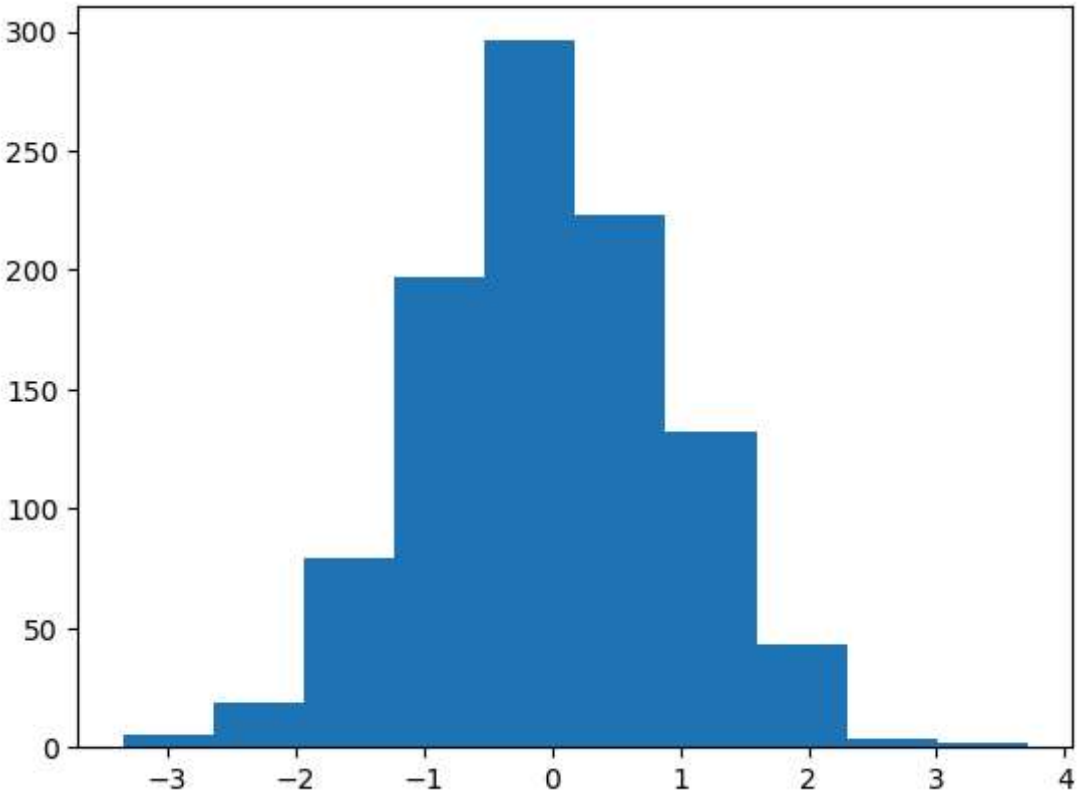


Histogram

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:-

In [22]:
```
data1 = np.random.randn(1000)

plt.hist(data1);
```

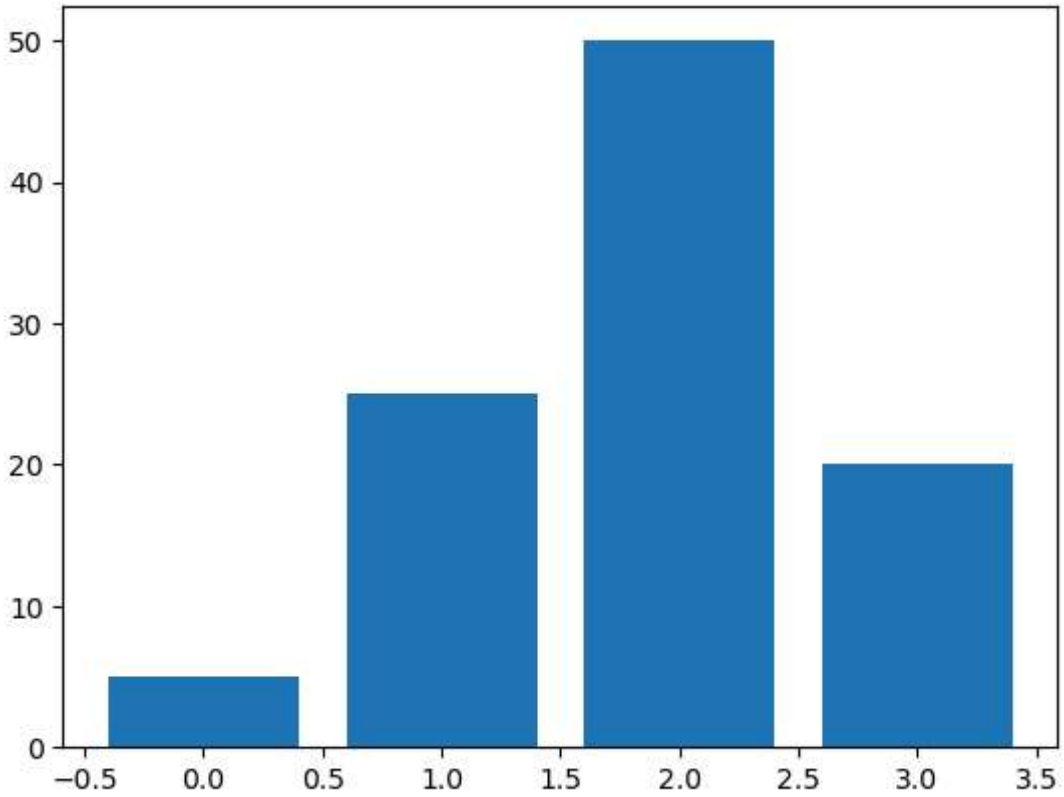


Bar Chart

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:-

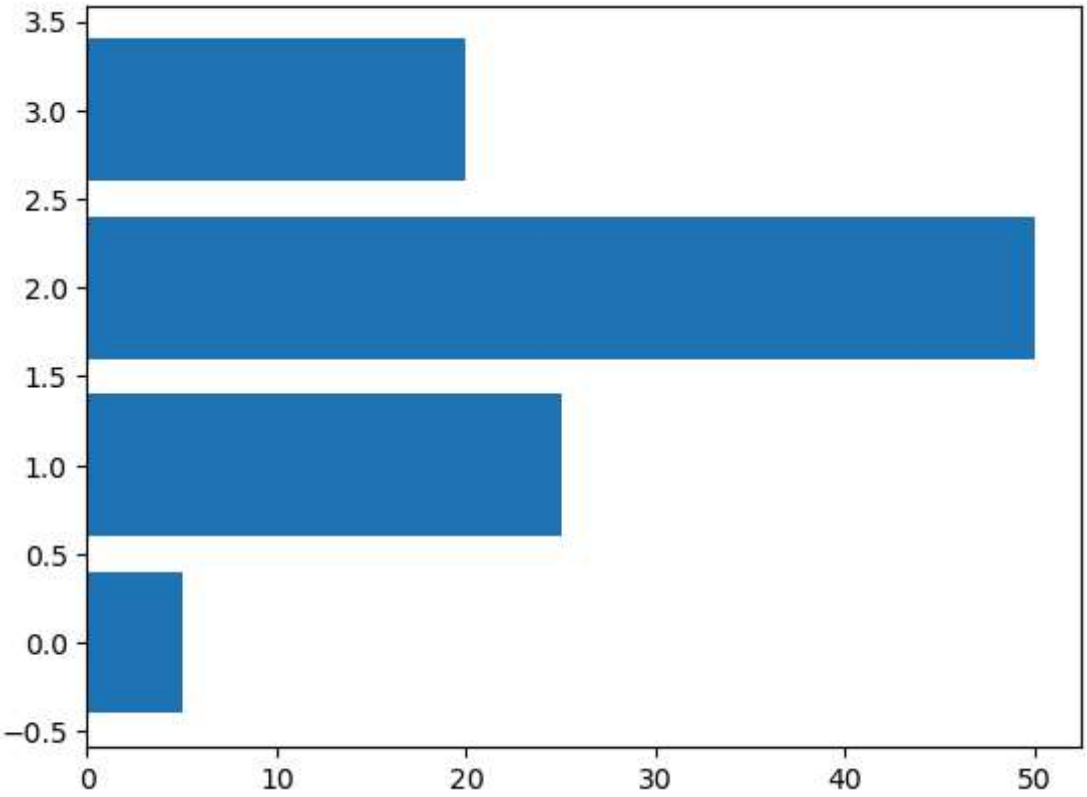
In [23]:
```
data2 = [5. , 25. , 50. , 20.]

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

plt.show()
```



Horizontal Bar Chart

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

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

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

plt.show()
```

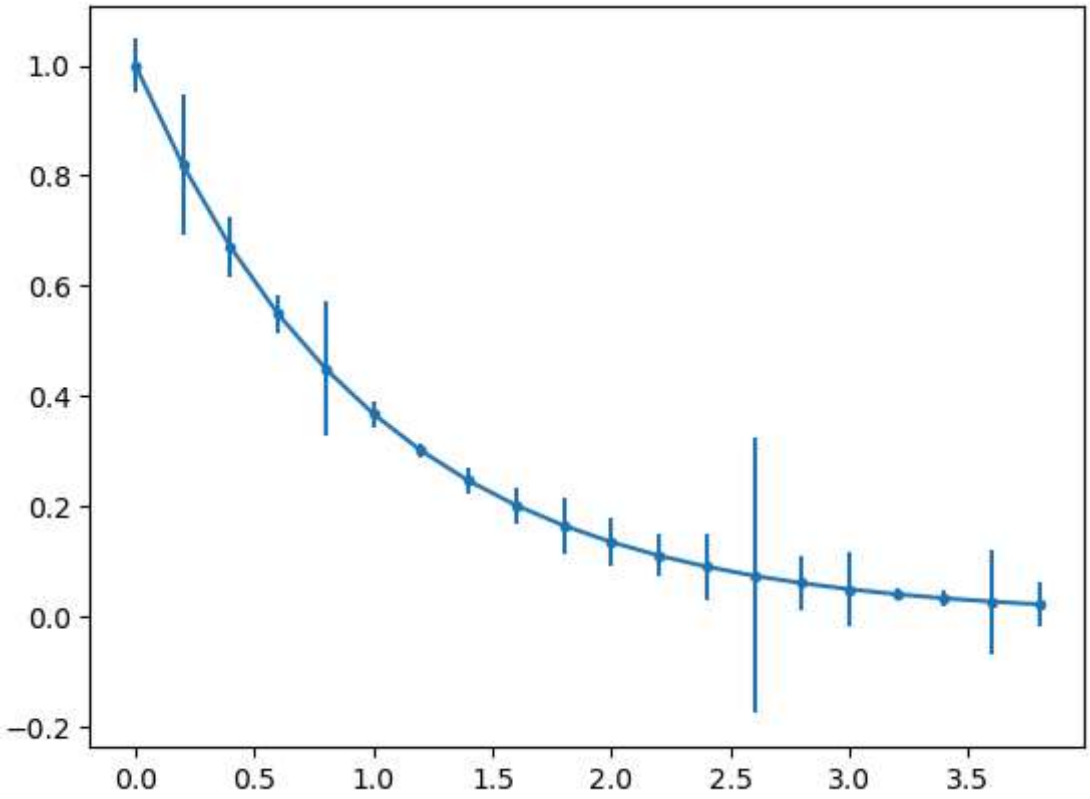


Error Bar Chart

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:-

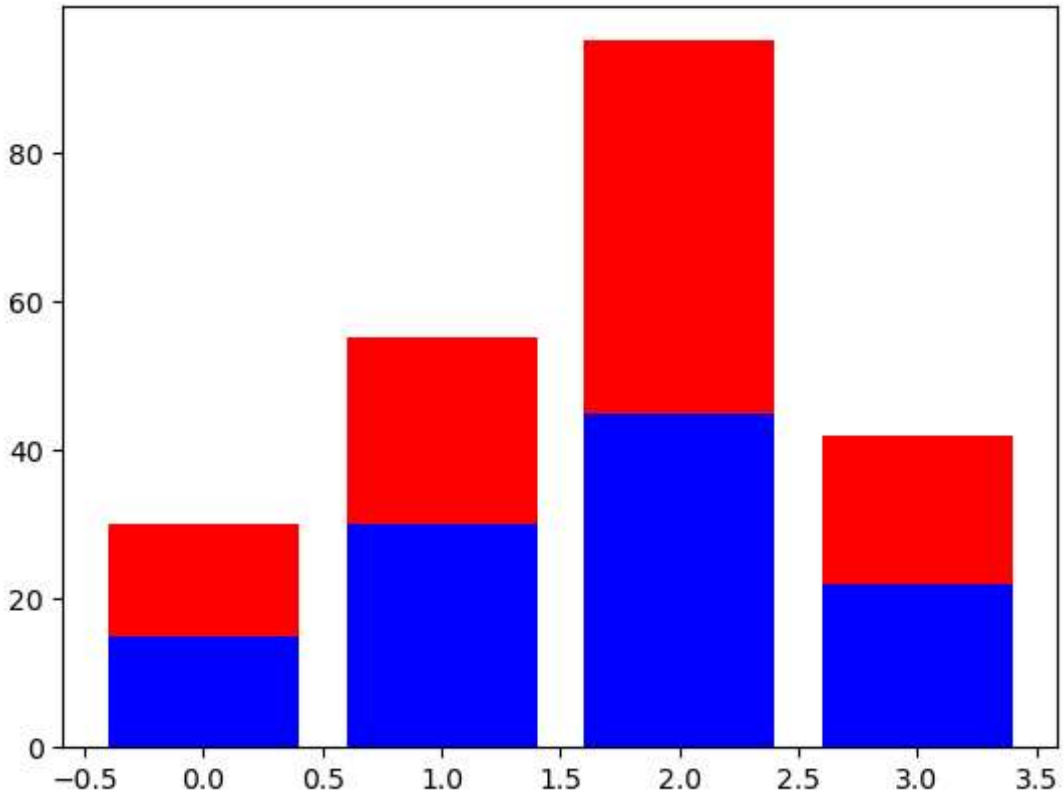
In [27]:
```
x9 = np.arange(0, 4, 0.2)

y9 = np.exp(-x9)

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

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

plt.show();
```



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:-

In [28]:
```
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()
```

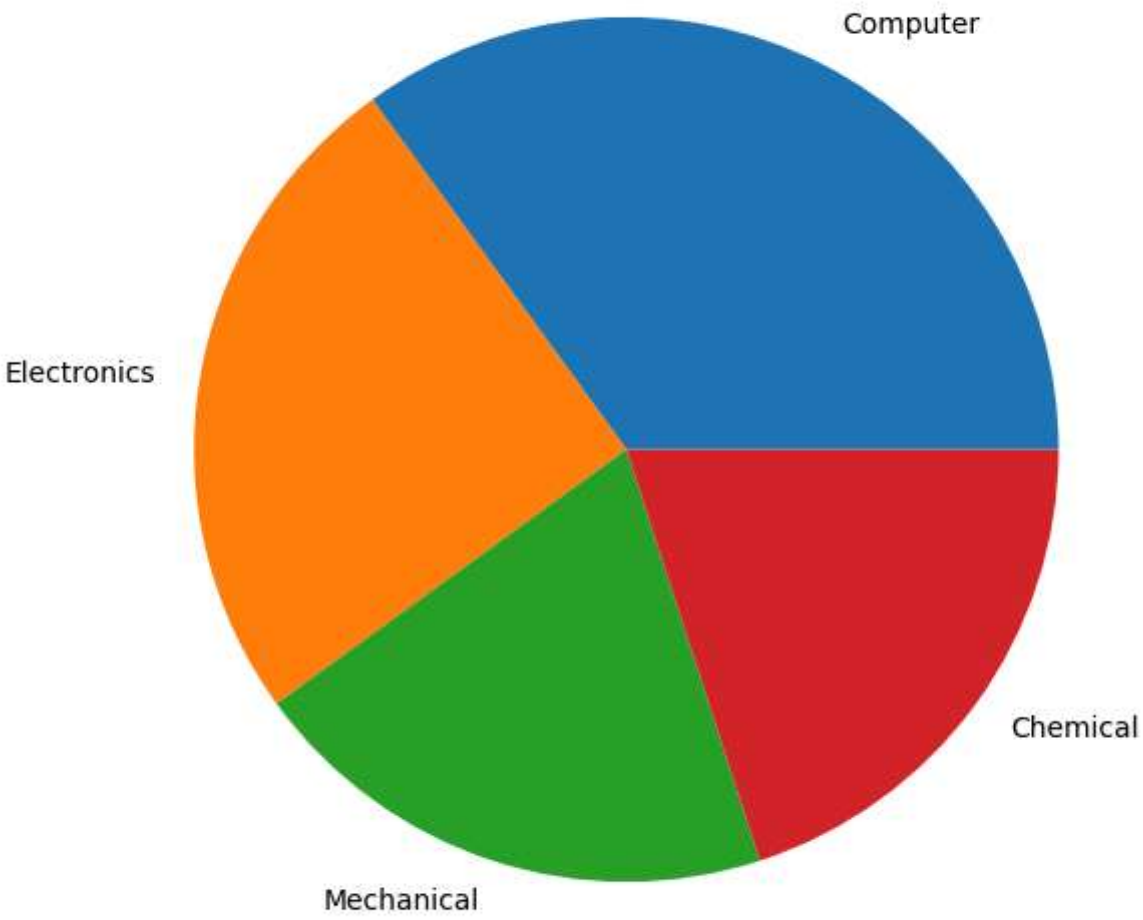


Pie Chart

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).

In [29]:
```
plt.figure(figsize=(7,7))

x10 = [35, 25, 20, 20]

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

plt.pie(x10, labels=labels);

plt.show()
```



Boxplot

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

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.

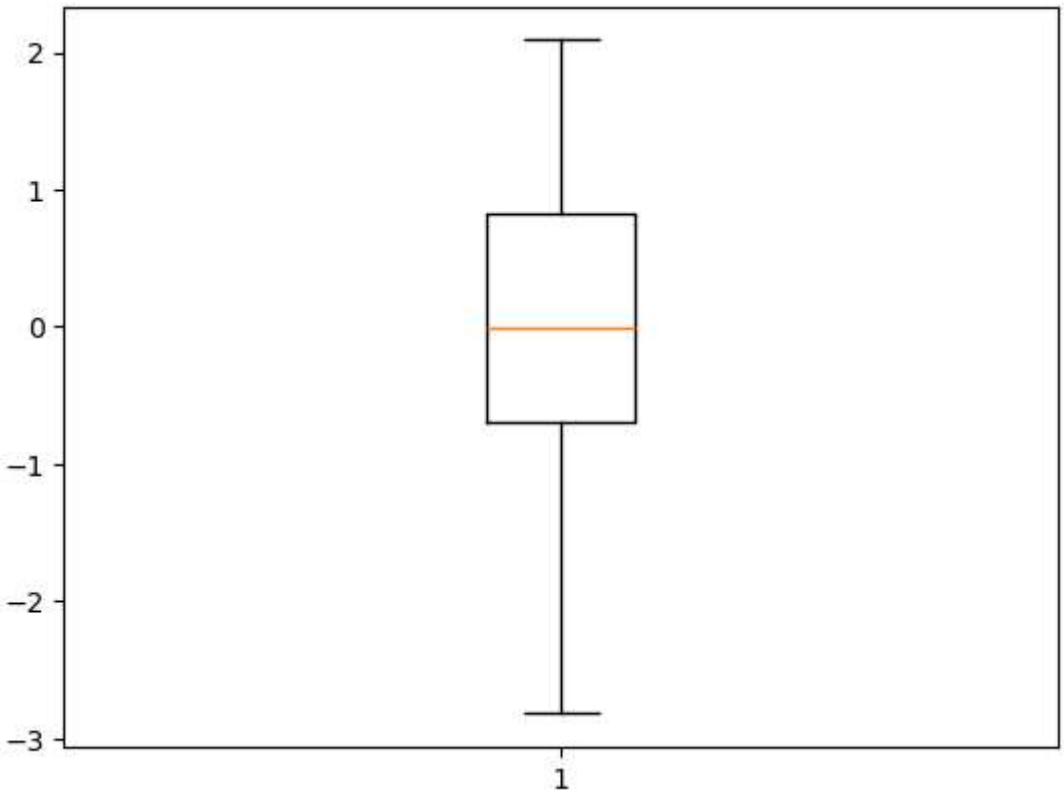
• The lower whisker extends to the lowest value within 1.5 IQR from the lower quartile.

• 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.

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

In [30]:
```
data3 = np.random.randn(100)

plt.boxplot(data3)

plt.show();
```

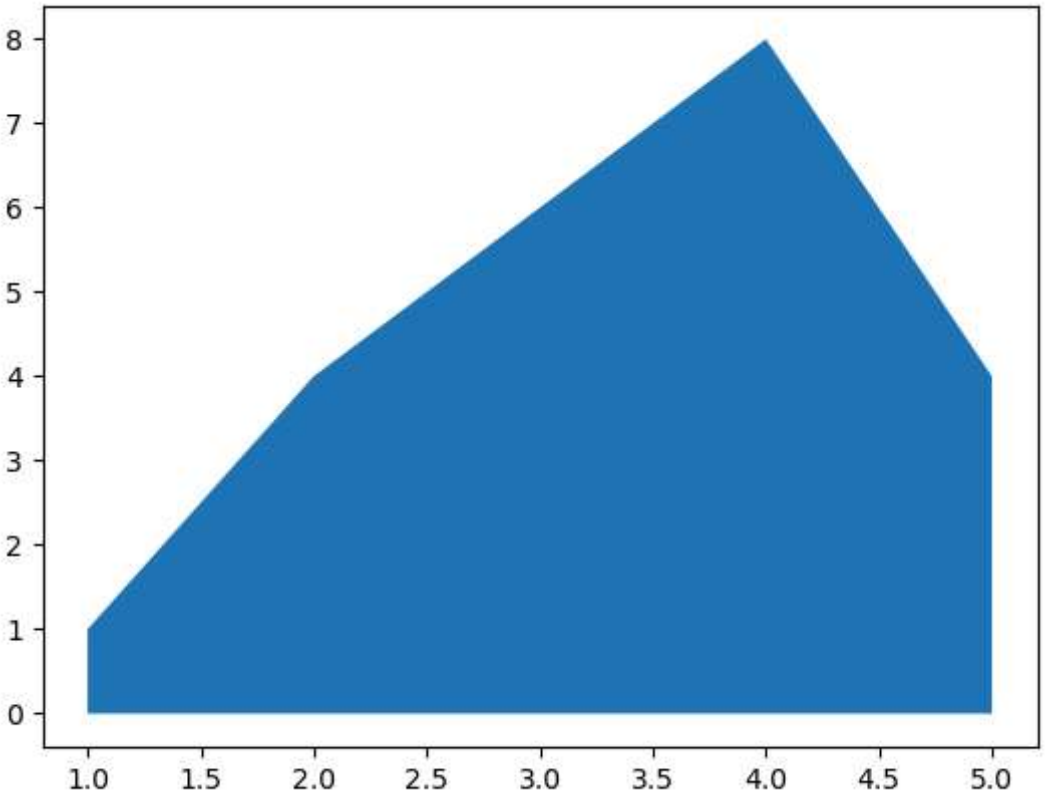


Area Chart

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:-

In [31]:
```
# Create some data
x12 = range(1, 6)
y12 = [1, 4, 6, 8, 4]

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



Contour Plot

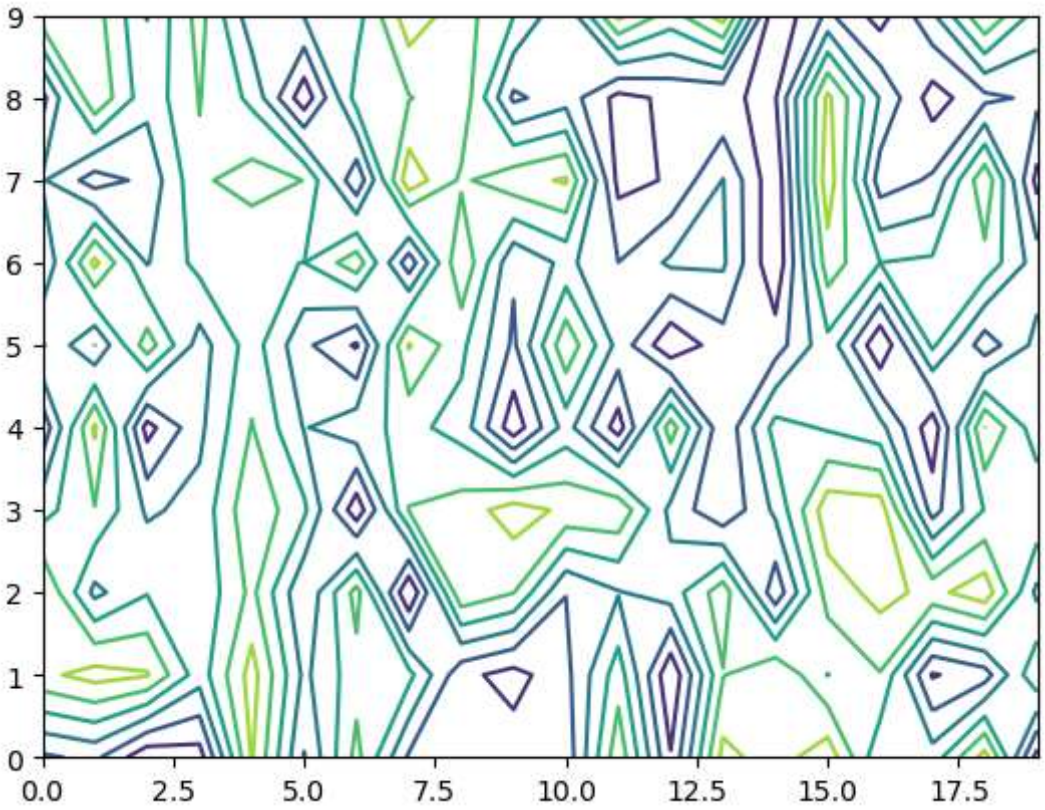
Contour plots are useful to display three-dimensional data in two dimensions using contours or color-coded regions. Contour lines are also known as level lines or isolines. Contour lines for a function of two variables are curves where the function has constant values. They have specific names beginning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

In [32]:
```
# 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)

In [33]:
```
# View list of all available styles

print(plt.style.available)
```

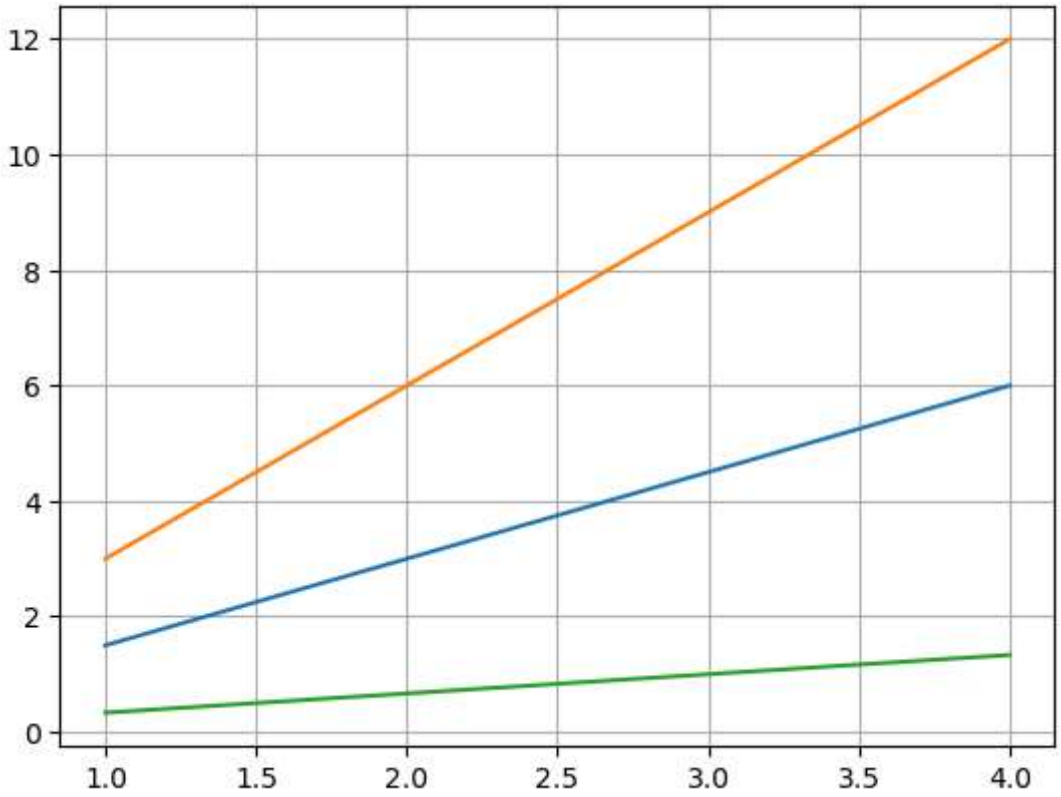
```
['Solarize_Light2', '_classic_test_patch', '_mpl-gallery', '_mpl-gallery-nogrid', 'bmh', 'classic', 'dark_background', 'fast', 'fivethirtyeight', 'ggplot', 'grayscale', 'seaborn-v0_8', 'seaborn-v0_8-bright', 'seaborn-v0_8-colorblind', 'seaborn-v0_8-dark', 'seaborn-v0_8-dark-palette', 'seaborn-v0_8-darkgrid', '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']
```

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.

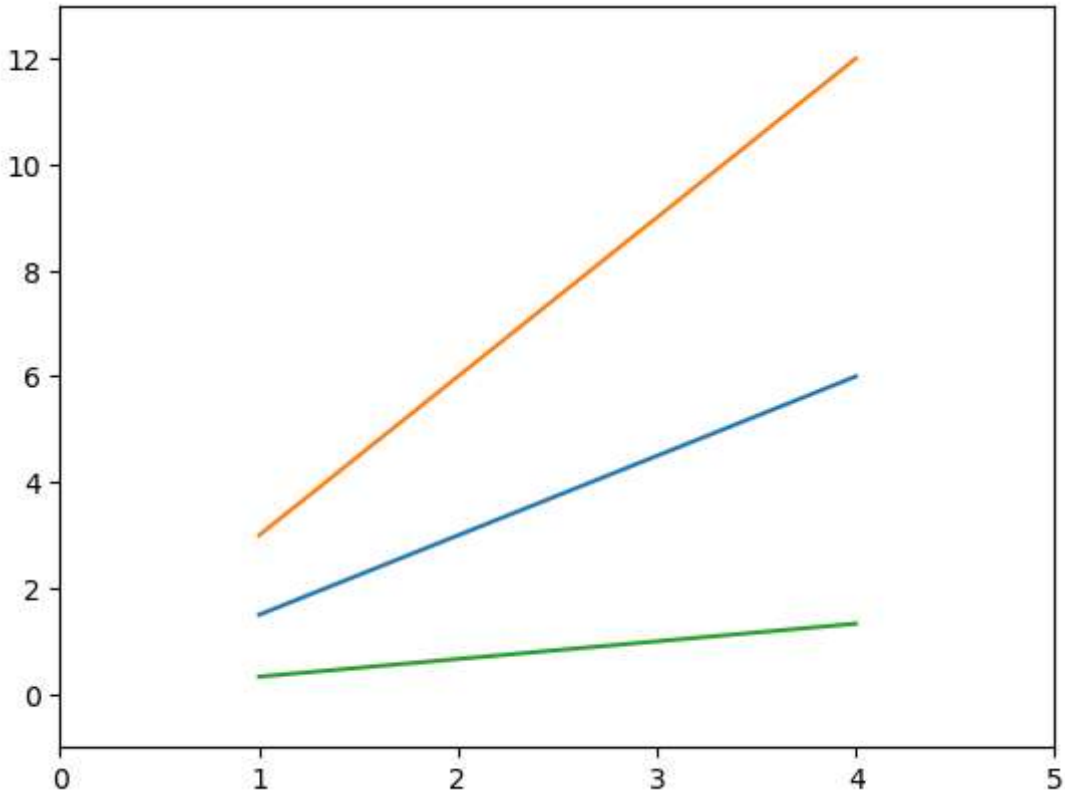
In [36]:
```
x15 = np.arange(1, 5)

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

plt.grid(True)

plt.show()
```



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:-

In [37]:
```
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.

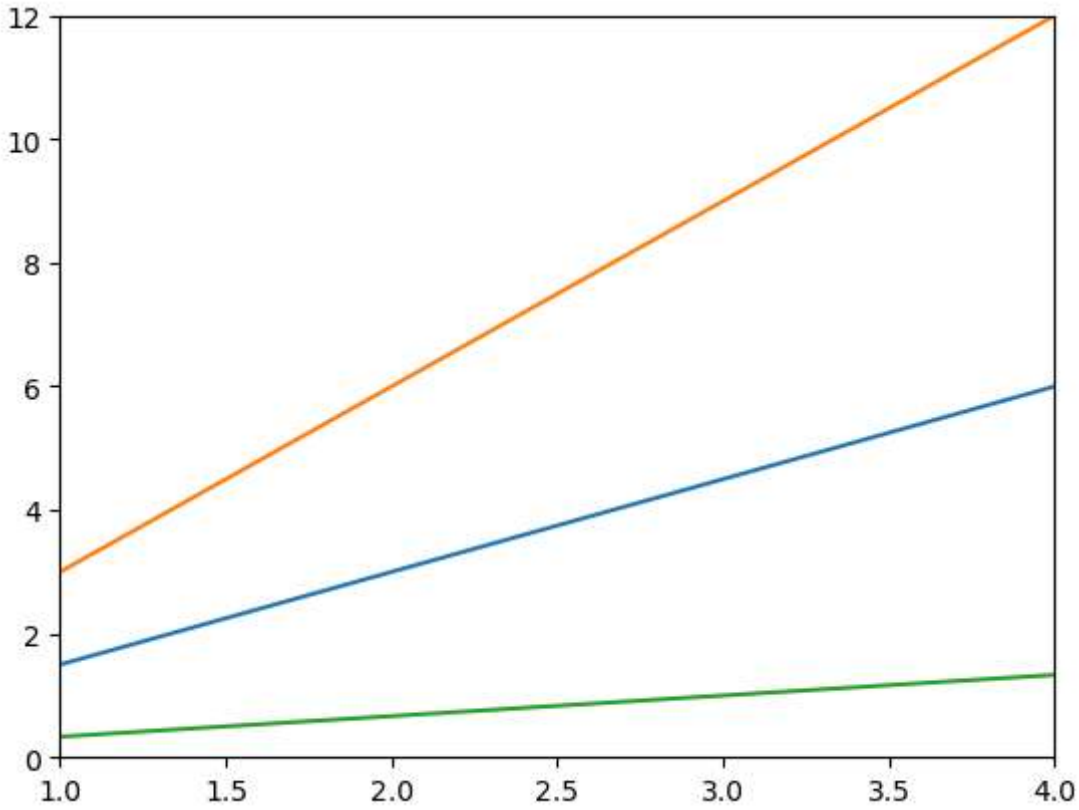
We can set parameters to axis() by a list of four values.

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:-

In [38]:
```
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])
```

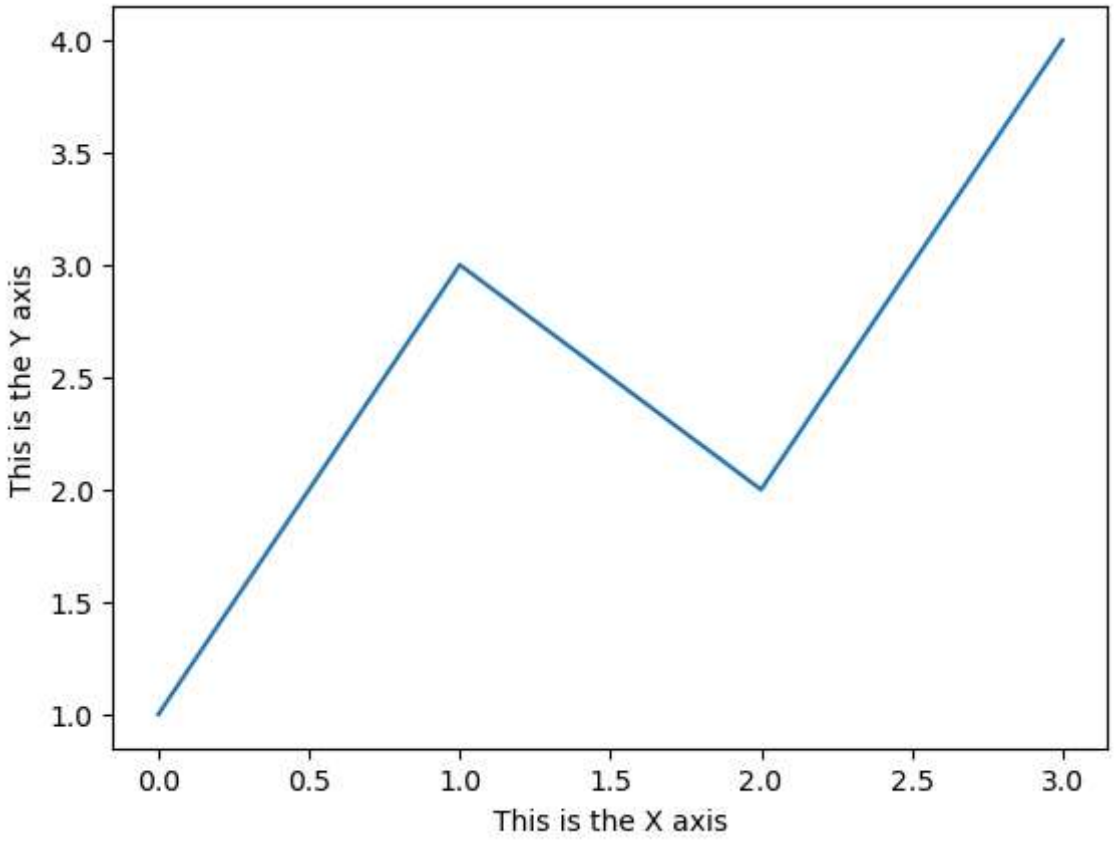
Out[38]:
```
(0.0, 12.0)
```



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.

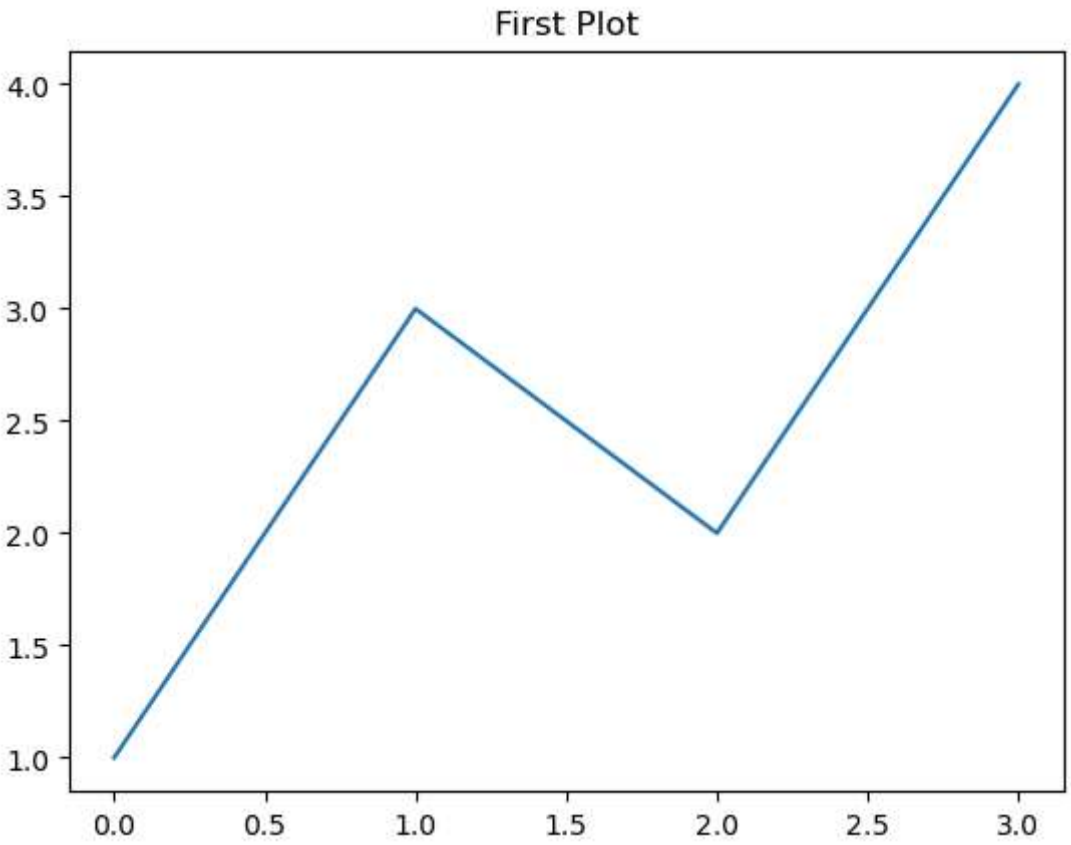
In [39]:
```
plt.plot([1, 3, 2, 4])

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

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

plt.show()
```



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.

In [40]:
```
plt.plot([1, 3, 2, 4])

plt.title('First Plot')

plt.show()
```

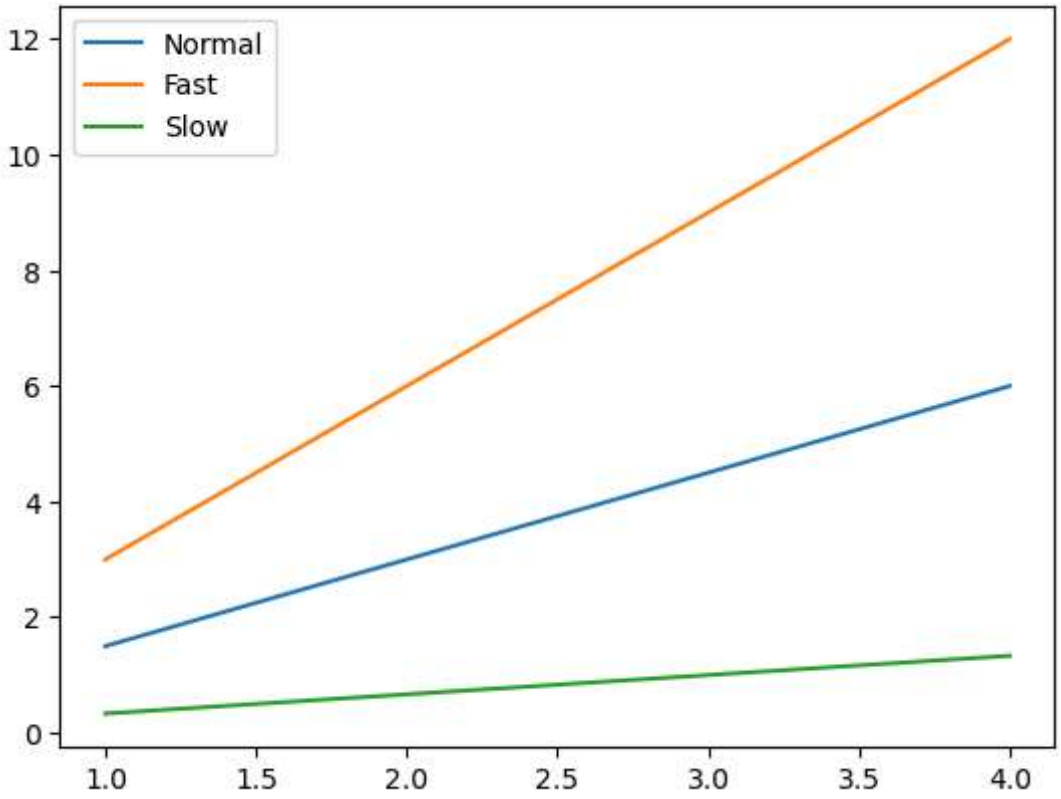


Adding a legend

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:-

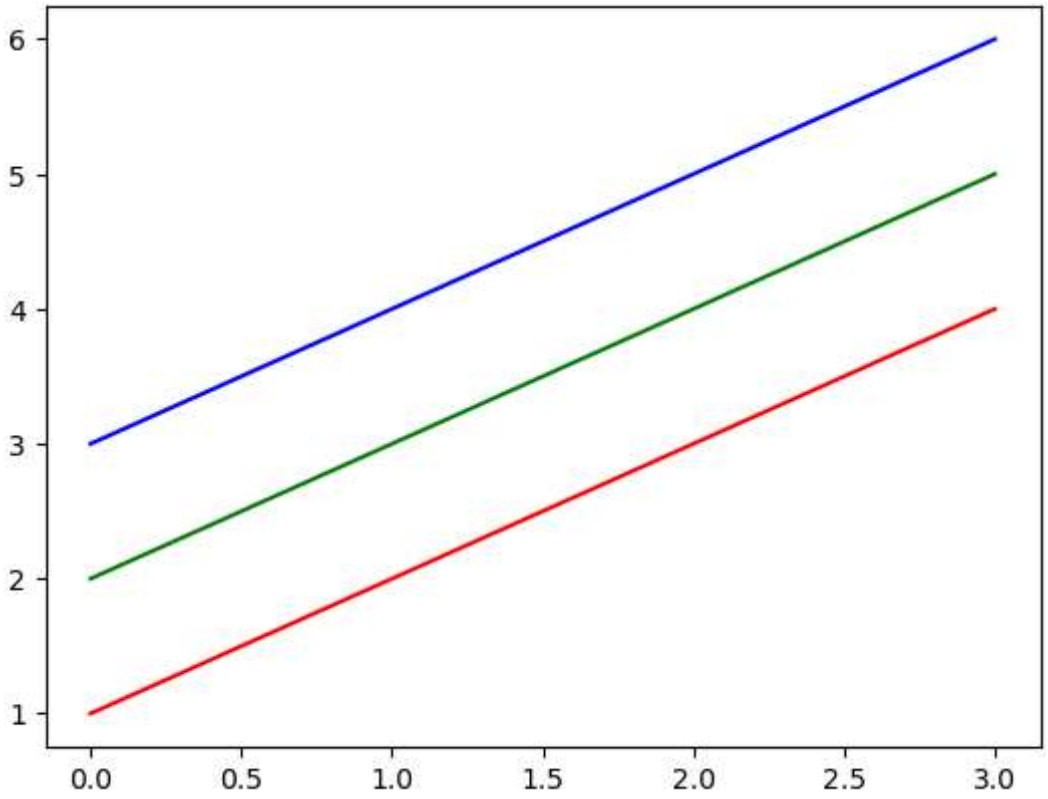
In [41]:
```
x15 = np.arange(1, 5)

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']);
```



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.

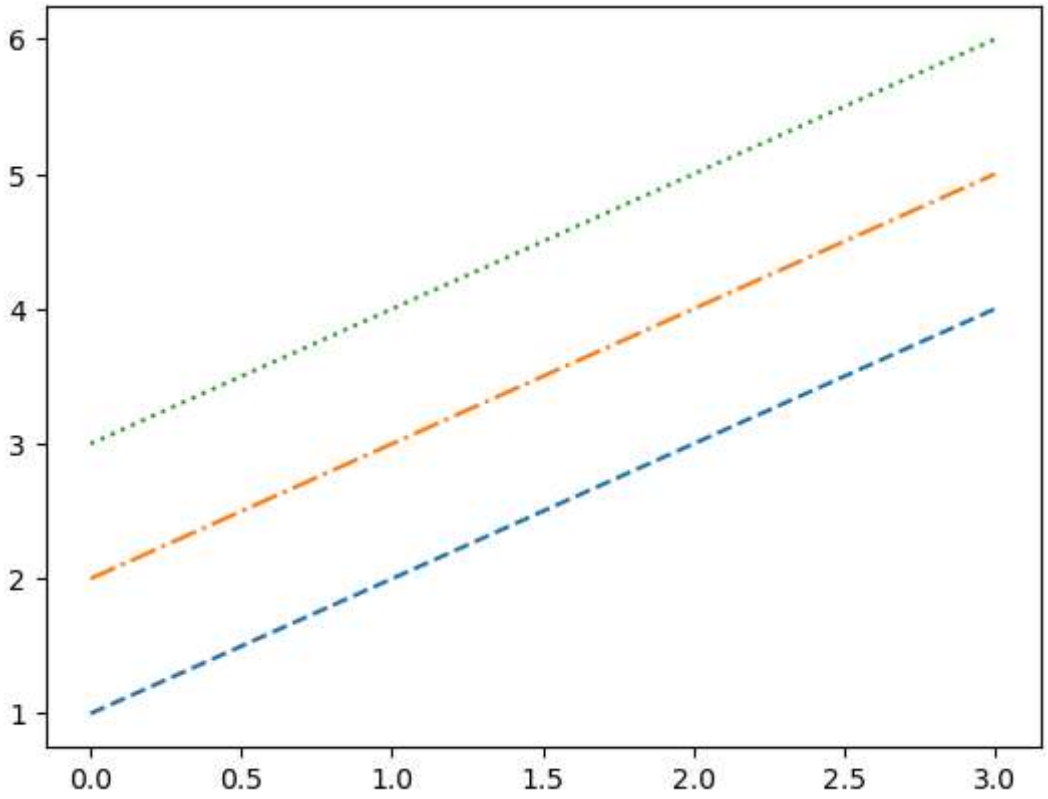
In [42]:
```
x16 = np.arange(1, 5)

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

plt.show()
```



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

In [43]:
```
x16 = np.arange(1, 5)

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

plt.show()
```



In [ ]: