Matplotlib

This project focuses on **Matplotlib**, a fundamental data visualization library in Python.

- **Matplotlib Object Hierarchy** — Understanding the building blocks of a plot.
- **Various Plot Types** — Line plots, bar charts, scatter plots, histograms, and more.
- **Customization Techniques** — Titles, labels, legends, colors, styles, and layouts.

The aim is to provide a **comprehensive and practical guide** to using Matplotlib effectively for visualizing data.

Table of Contents

1. Introduction

When we want to share information, there are many ways to do it.
One of the most effective ways is **Data Visualization** — showing data using **plots and graphics**.

In data visualization:

- We take **numerical data** as input.
- We display it as **charts, figures, or tables**.
- This makes complex data **easier to understand** and helps in making **better decisions.**

The goal of data visualization is to **communicate information clearly and effectively.**

In this project, we will focus on **Matplotlib**, the most basic and widely used data visualization tool in Python.
Before we dive into Matplotlib, let's take a quick look at other Python data visualization tools.

---

2. Overview of Python Data Visualization Tools

Python is one of the most popular programming languages for **data science**.
It offers many libraries for visualizing data effectively.

Some popular Python data visualization tools include:

- **Matplotlib**
- **Seaborn**
- **pandas** (built-in plotting)
- **Bokeh**
- **Plotly** (also available in R)
- **ggplot** (R package)
- **Pygal** (Python & R support)

In the following sections, we will explore **Matplotlib** in detail.

---

3. Introduction to Matplotlib

Matplotlib is the **core plotting library** in Python.
It is highly versatile and can create a wide range of visualizations, including:

- Line plots

- Scatter plots
- Histograms
- Bar charts (vertical & horizontal)
- Error bar charts
- Pie charts
- Box plots
- Area charts
- Contour plots
- 3D plots

It can save plots in formats like **PDF, SVG, JPG, PNG, BMP, and GIF.**
Other libraries like **pandas** and **Seaborn** are built on top of Matplotlib.

History:

- Created by **John Hunter** in 2002 to visualize **ECoG data** for epilepsy research.
- Became open-source and widely adopted.
- Even used in NASA's **Phoenix spacecraft landing in 2008** for data visualization.

---

4. Importing Matplotlib

To start using Matplotlib, we first need to import it:

```
import matplotlib
```

```
Most of the time, we use pyplot, a simpler interface for plotting:

import matplotlib.pyplot as plt

We often also import NumPy and pandas for handling data:

# Import dependencies
import numpy as np
import pandas as pd

# Import Matplotlib
import matplotlib.pyplot as plt
5. Displaying Plots in Matplotlib
The way we display plots depends on how we are running Python code.
There are three common cases:

5.1 Plotting from a Script
Use plt.show() at the end of the script to display all figures.

Avoid calling plt.show() multiple times – it may cause issues.

5.2 Plotting from an IPython Shell
Enable interactive plotting with:

%matplotlib
Any plotting command will open a figure window that can be updated.
```

5.3 Plotting from a Jupyter Notebook
Jupyter Notebook combines code, visualizations, text, and media in one document.

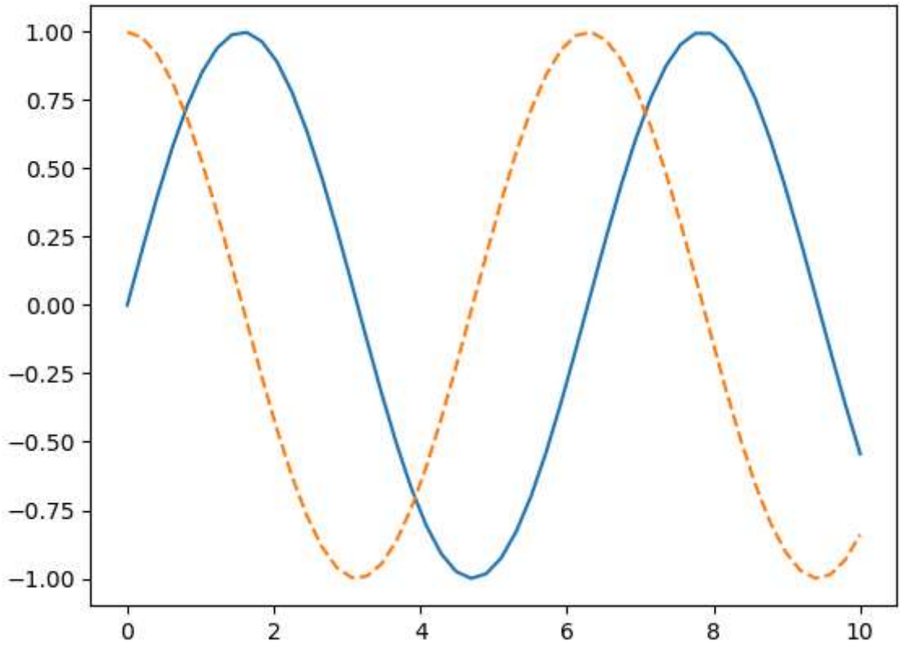
To enable plotting in Jupyter:

%matplotlib inline   # For static plots inside the notebook
%matplotlib notebook # For interactive plots inside the notebook
Run this command once per session.
After that, any plot will appear directly inside the notebook.

This version:
- Uses simple, conversational language.
- Keeps section numbers for organization.
- Formats code examples cleanly for Jupyter.
- Splits into short paragraphs for better readability.

If you want, I can also add clickable navigation links so each heading in this text works lik this text works like an interactive Table of Contents in your Jupyter Notebook.
That would make it feel like a professional documentation page.

In [2]:
```
import matplotlib # import the matplotlib library
import matplotlib.pyplot # import the pyplot module
import matplotlib.pyplot as plt # import the pyplot module as plt
import numpy as np # import the numpy library
import pandas as pd
x1 = np.linspace(0, 10, 50) # create an array of 50 values between 0 and 10

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

plt.plot(x1, np.sin(x1), '-') # plot the sine function
plt.plot(x1, np.cos(x1), '--') # plot the cosine function
#plt.plot(x1, np.tan(x1), '--') # plot the tangent function
plt.show() # show the plot
```



6. Matplotlib Object Hierarchy

Matplotlib plots are built using a **hierarchy of Python objects** — think of it like a tree structure.

- **Figure** → The outermost container (the whole visualization window).
- **Axes** → Each Figure can have one or more Axes (individual plots).
- **Axis** → The x-axis and y-axis inside an Axes.
- **Other elements** → Tick marks, lines, legends, titles, text boxes, etc.

You can imagine the **Figure** as a big box, and inside it are one or more **Axes.**
Each Axes contains the actual plot and all the smaller elements.

---

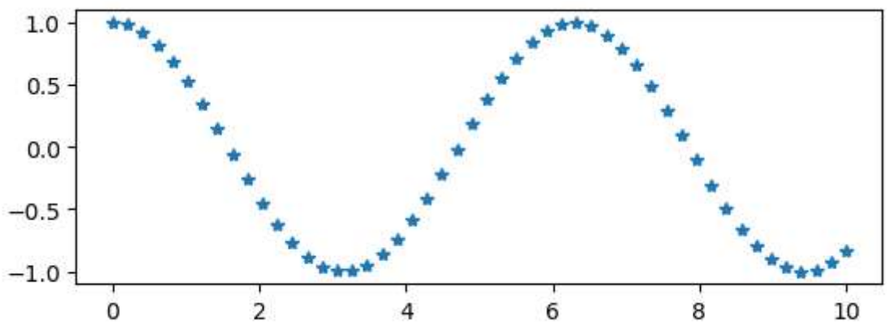
7. Matplotlib API Overview

Matplotlib provides two main ways (APIs) to create plots:

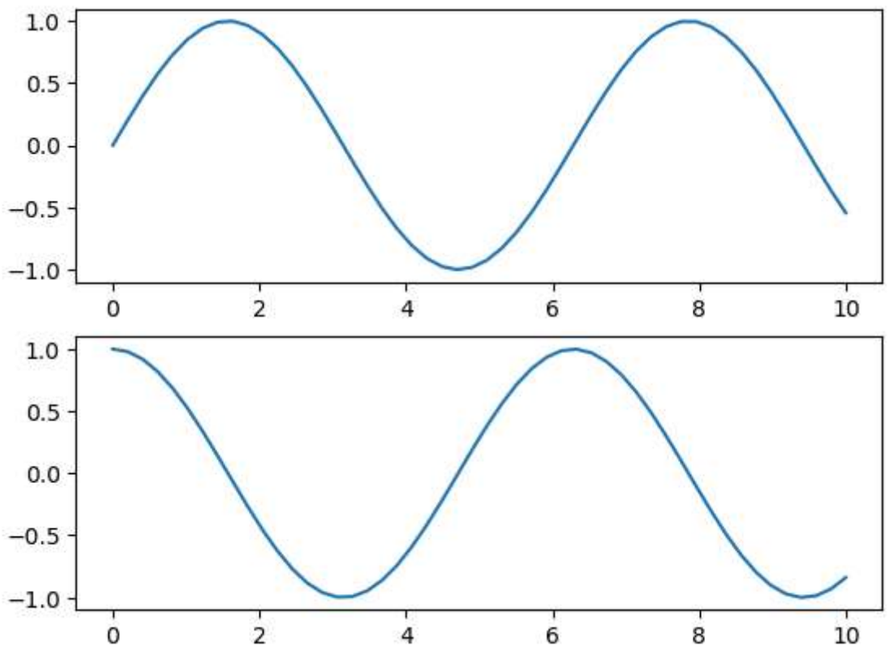
1. **Pyplot API (MATLAB-style)**
   - Easy to use for quick, interactive plots.
   - Works like MATLAB commands (`plt.plot()` , `plt.show()`).
2. **Object-Oriented (OO) API**
   - Gives more control and flexibility.
   - Better for complex plots or when working with multiple figures.

> There's also an older **pylab interface** (combines Pyplot + NumPy), but it's

In [3]:
```
# create the first of two panels and set current axis
plt.subplot(2, 1, 1)   # (rows, columns, panel number)
plt.plot(x1, np.cos(x1), '*') # plot the cosine function
plt.show() # show the plot
```



In [4]:
```
# create a plot figure
plt.figure() # create a 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)) # plot the sine function

# 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)); # plot the cosine function
plt.show()
```



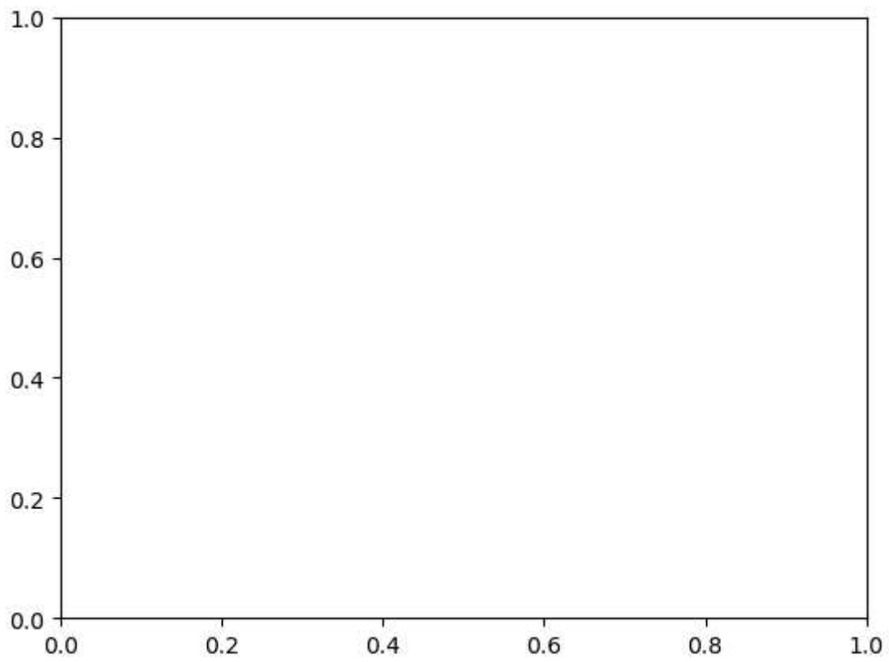
In [5]:
```
# get current figure information
```

```
print(plt.gcf()) # print the current figure
```

```
Figure(640x480)
<Figure size 640x480 with 0 Axes>
```

In [6]:
```
# get current axis information

print(plt.gca()) # print the current axis
plt.show()
```

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



Visualization with Pyplot

Creating a plot with **Pyplot** is simple.
In this example:

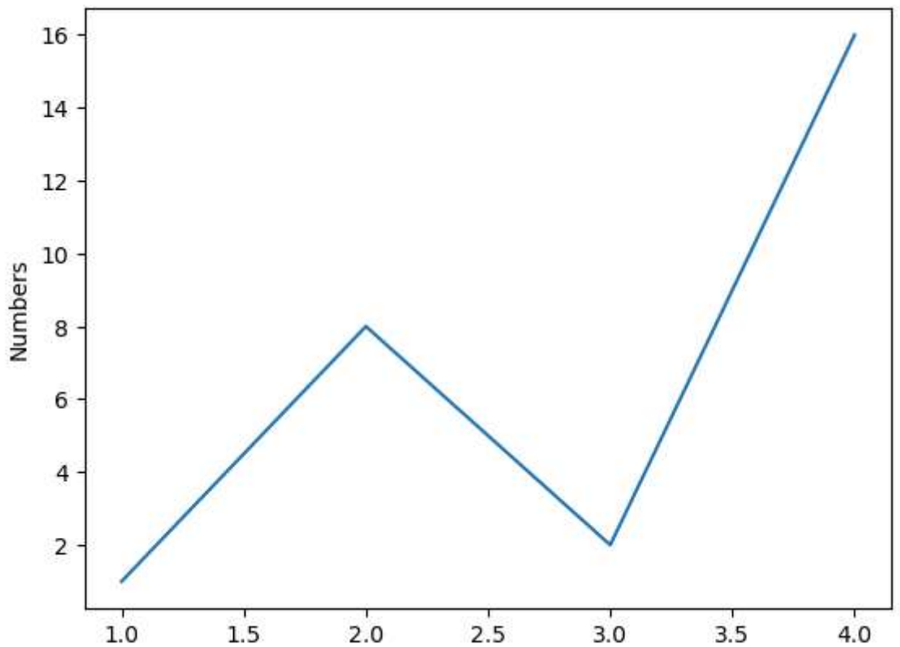
- The **x-axis** values go from `0` to `3`.
- The **y-axis** values go from `1` to `4`.

If we pass only a single list or array to the `plot()` function, Matplotlib assumes these are **y-values.**
It then automatically creates **x-values** starting from `0`, with the same length as the y-values.

For example:

- **y-values** = `[1, 2, 3, 4]`
- **x-values** (generated automatically) = `[0, 1, 2, 3]`

In [7]:
```
plt.plot([1,2,3,4], [1,8,2,16]) # plot the data
plt.ylabel('Numbers') # label the y-axis
plt.show() # show the plot
```



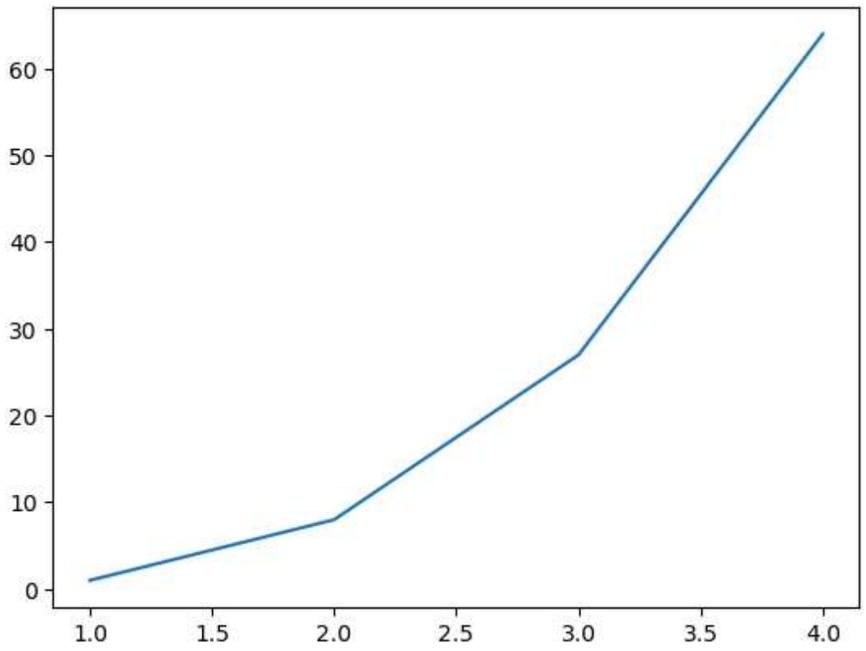
`plot()` – A Versatile Command

The `plot()` function in Matplotlib is very flexible.
It can take multiple arguments to create different types of plots.

For example, to plot **x** values against **y** values, you can use:

```
plt.plot(x, y)
```

In [8]:
```
plt.plot([1, 2, 3, 4], [1, 8, 27, 64]) # plot the data
plt.show() # show the plot
```



State-Machine Interface

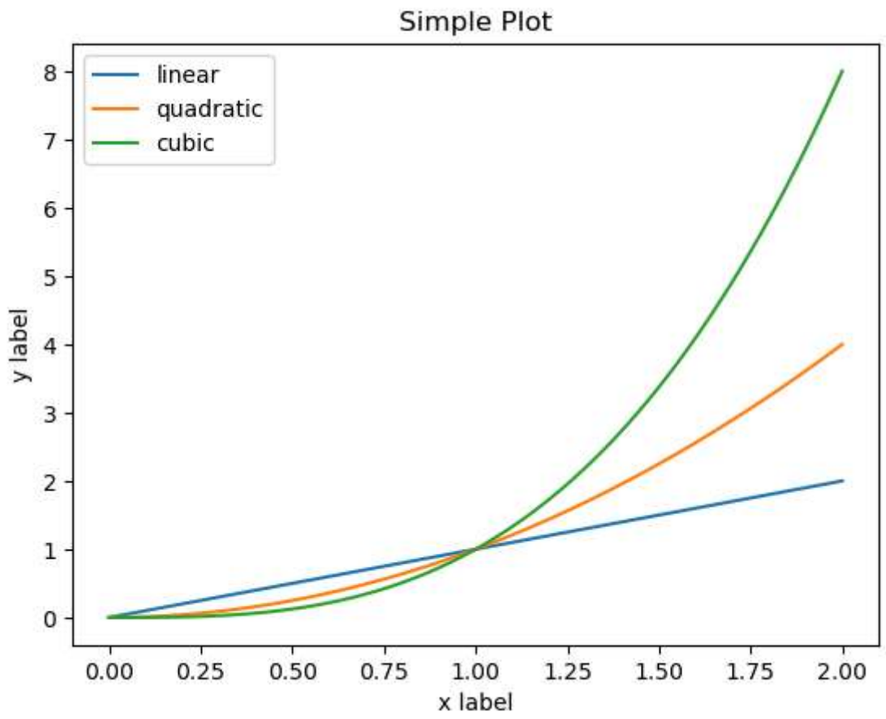
Pyplot uses a *state-machine* style interface for plotting.

This means it automatically creates the **figure** and **axes** you need, without you having to set them up manually.

You just call plotting functions, and Pyplot takes care of creating and updating the plot in the background.

Example:

In [9]:
```
x = np.linspace(0, 2, 100) # create an array of 100 values between 0 and 2

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

plt.xlabel('x label') # label the x-axis
plt.ylabel('y label') # label the y-axis

plt.title("Simple Plot") # title the plot

plt.legend() # show the legend

plt.show() # show the plot
```



Formatting the Style of a Plot

When plotting in Matplotlib, each pair of `x` and `y` values can have an **optional third argument** called the **format string**.
This format string tells Matplotlib **the color, marker style, and line style** for the plot.

The format codes come from **MATLAB-style notation**.
You can combine:

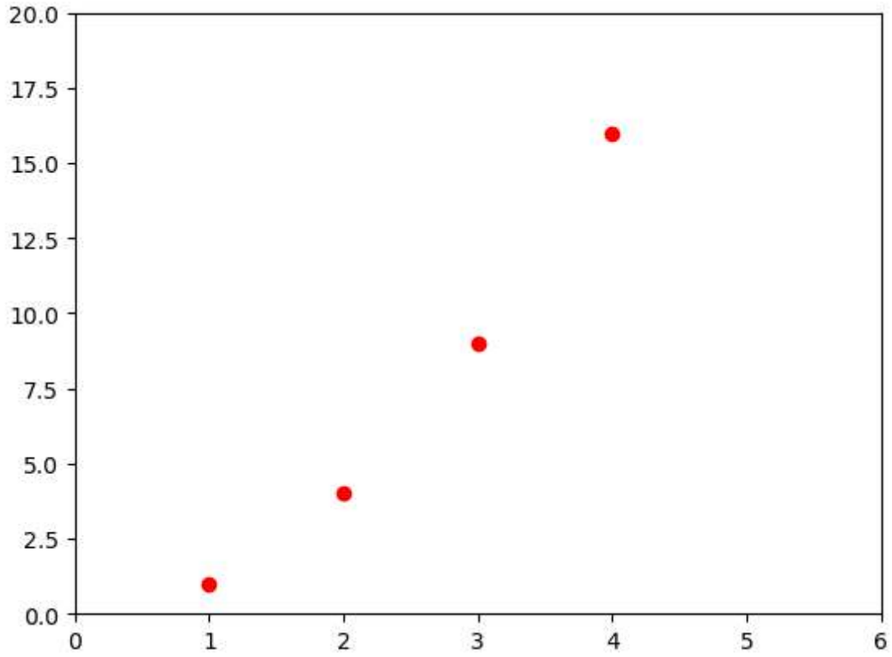
- A **color code** (e.g., `r` for red, `b` for blue)
- A **marker symbol** (e.g., `o` for circle, `s` for square)
- A **line style** (e.g., `-` for solid line, `--` for dashed line)

Default style: `'b-'` → solid blue line.

Example:
To plot a line with **red circles**, use:

```
plt.plot(x, y, 'ro')
```

In [10]:
```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro') # plot the data with circles
plt.axis([0, 6, 0, 20]) # set the axis limits
plt.show() # show the plot
```



The `axis()` command sets the visible range of the plot.
It takes a list in the format: `[xmin, xmax, ymin, ymax]` — where:

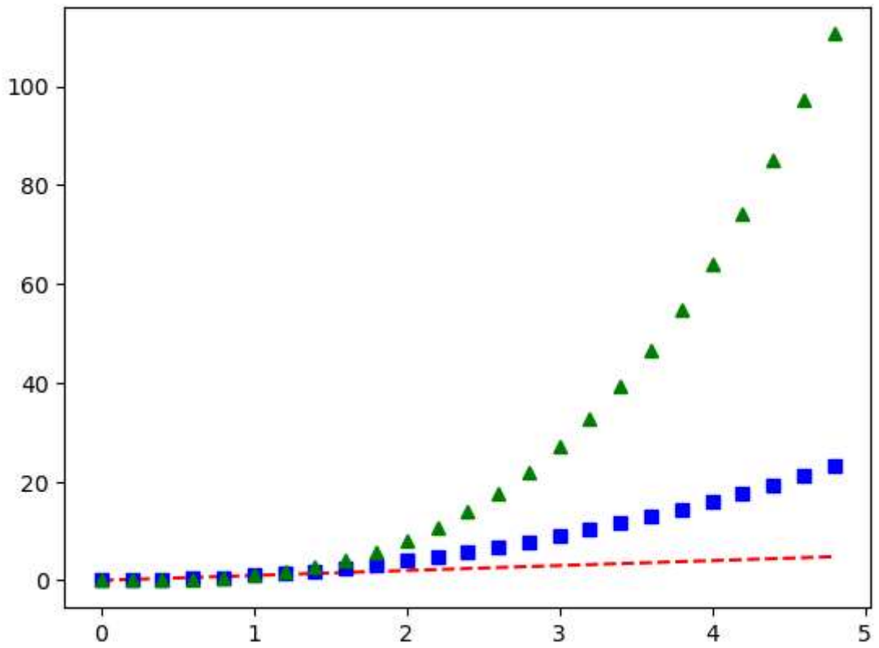
- **xmin** and **xmax** set the horizontal range
- **ymin** and **ymax** set the vertical range

Working with NumPy Arrays

In Matplotlib, data is often handled as **NumPy arrays.**
Even if you pass in Python lists or other sequences, Matplotlib automatically converts them into NumPy arrays.

The example below shows how to plot **multiple lines** with different styles in a single command using arrays.

In [11]:
```
# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2) # create an array of 25 values between 0 and 5

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^') # plot the data with different styles for each line (red dashes, blue squares, green triangles)

plt.show() # show the plot
```



9. Object-Oriented API

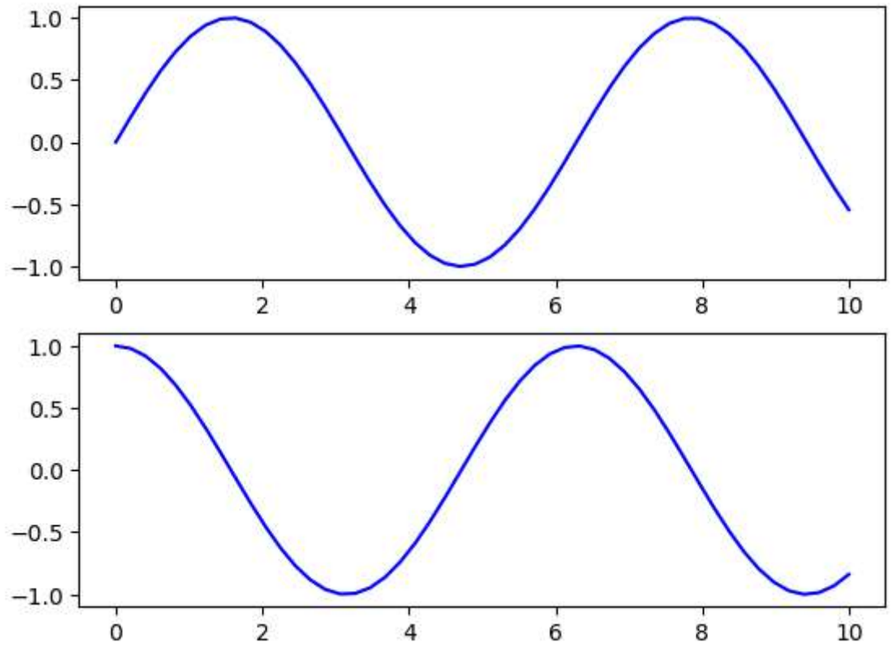
The **Object-Oriented (OO) API** in Matplotlib is used for more **advanced and flexible plotting**.
It gives us **full control** over the figure and its elements.

In the **Pyplot API**, we rely on an "active" figure or axes, and plotting functions are called directly.
But in the **OO API**, plotting functions are called as **methods** of **explicit** `Figure` and `Axes` objects.

- **Figure**: The top-level container that holds everything in a plot. Think of it as a canvas or box that contains one or more plots.
- **Axes**: An individual plot within a figure. Each `Axes` object contains smaller elements such as:
  - Axis (X and Y)
  - Tick marks
  - Lines
  - Legends
  - Title
  - Text boxes

Example: The following code creates sine and cosine curves using the OO API.

In [12]:
```
# First create a grid of plots
# ax will be an array of two Axes objects
fig, ax = plt.subplots(2) # create a figure with two axes


# Call plot() method on the appropriate object
ax[0].plot(x1, np.sin(x1), 'b-') # plot the sine function on the first axis
ax[1].plot(x1, np.cos(x1), 'b-'); # plot the cosine function on the second axis

plt.show()
```



Objects and References in Matplotlib

In the **Object-Oriented API** of Matplotlib, we work with **objects** (like figures and axes) and then **apply methods** (functions) to them.

The main benefit of this approach appears when:

- You create **multiple figures** in the same code.
- A single figure contains **multiple subplots**.

Here's how it works:

1. First, we create a **Figure object** and store it in a variable, for example, `fig`.
2. Then, we create an **Axes object** (the actual plotting area) using the `add_axes()` method of the `Figure` object.

Example:

```
fig = plt.figure()          # Create a Figure
axes = fig.add_axes([0, 0, 1, 1])  # Create Axes inside the Figure
```

In [13]:
```
fig = plt.figure() # create a figure

x2 = np.linspace(0, 5, 10) # create an array of 10 values between 0 and 5
y2 = x2 ** 2 # create an array of 10 values between 0 and 5

axes = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # add an axes to the figure

axes.plot(x2, y2, 'r') # plot the data with red color

axes.set_xlabel('x2') # set the x-axis label
axes.set_ylabel('y2') # set the y-axis label
axes.set_title('title') # set the title
plt.show() # show the plot
```

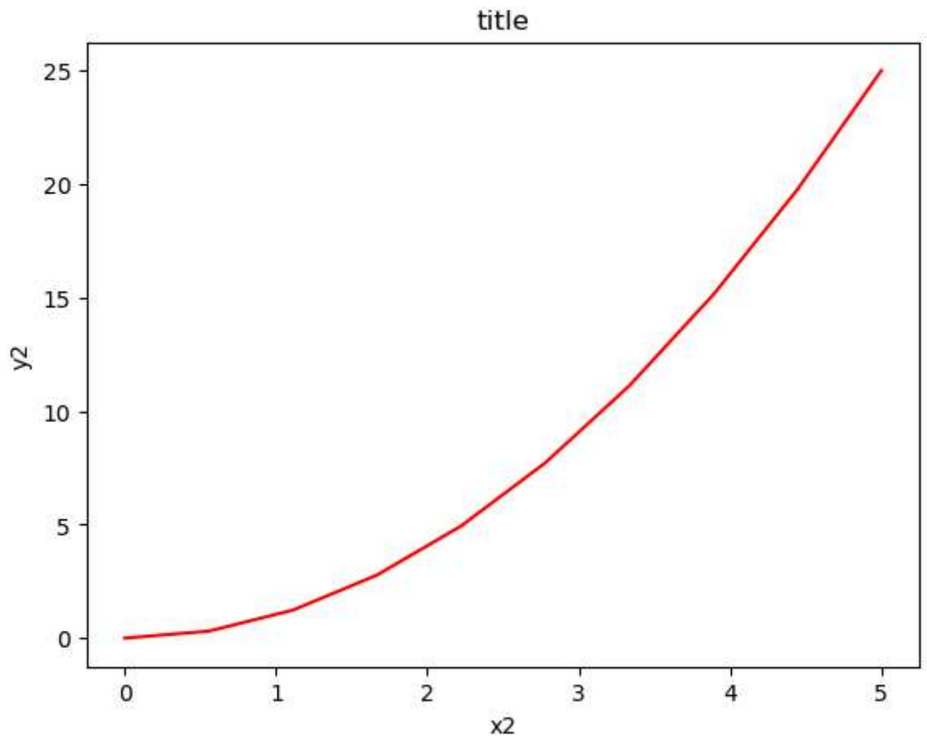


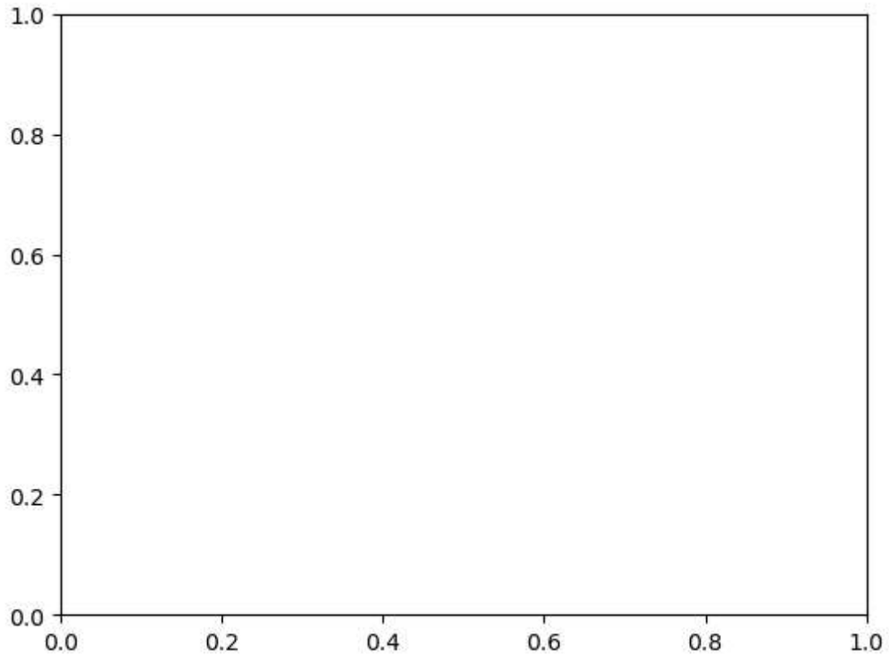
Figure and Axes

In Matplotlib, a **Figure** is the main container that holds everything you see in a plot — axes, graphics, text, and labels. An **Axes** is the actual area where your data gets plotted. It includes the x-axis, y-axis, ticks, and labels.

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

- **Figure (fig) → The entire plotting canvas.**
- **Axes (ax) → The specific plot area inside the figure.**
- The Figure is the whole canvas.
- The Axes is the frame where the picture (your data) is drawn.

In [14]:
```
fig = plt.figure() # create a figure
ax = plt.axes() # create an axes
plt.show() # show the plot
```



10. Figure and Subplots

In Matplotlib, all plots live inside a **Figure** object.

We can create a new figure using:

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

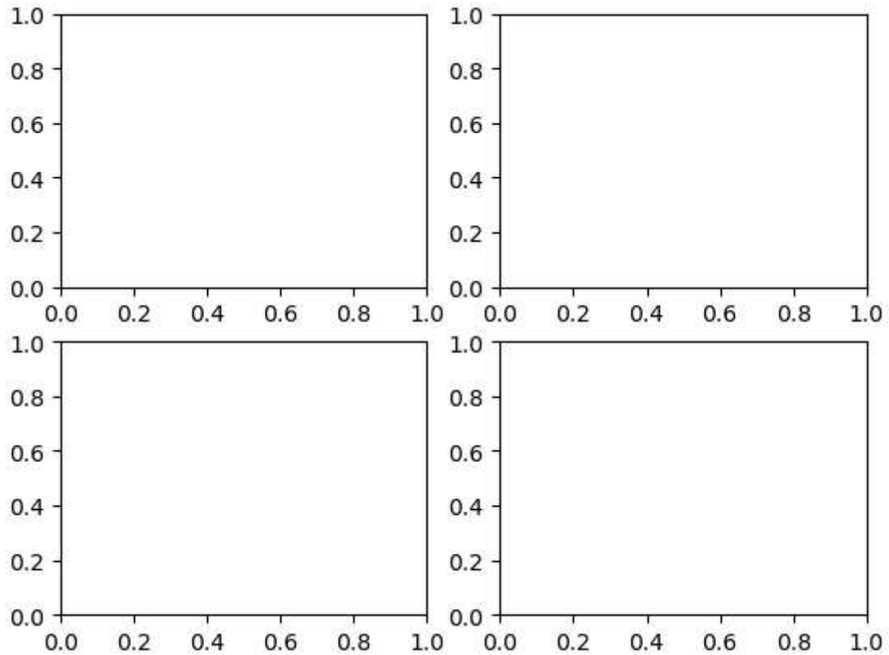
- A subplot is a smaller plot inside the figure.
- We can add one or more subplots with **fig.add_subplot():**

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

- The first two numbers (2, 2) mean the figure is divided into **2 rows** and **2 columns**, making **4 subplots** in total.
- The last number 1 means we are selecting the **first subplot** (counting starts from 1).

```
ax2 = fig.add_subplot(2, 2, 2)
ax3 = fig.add_subplot(2, 2, 3)
ax4 = fig.add_subplot(2, 2, 4)
```

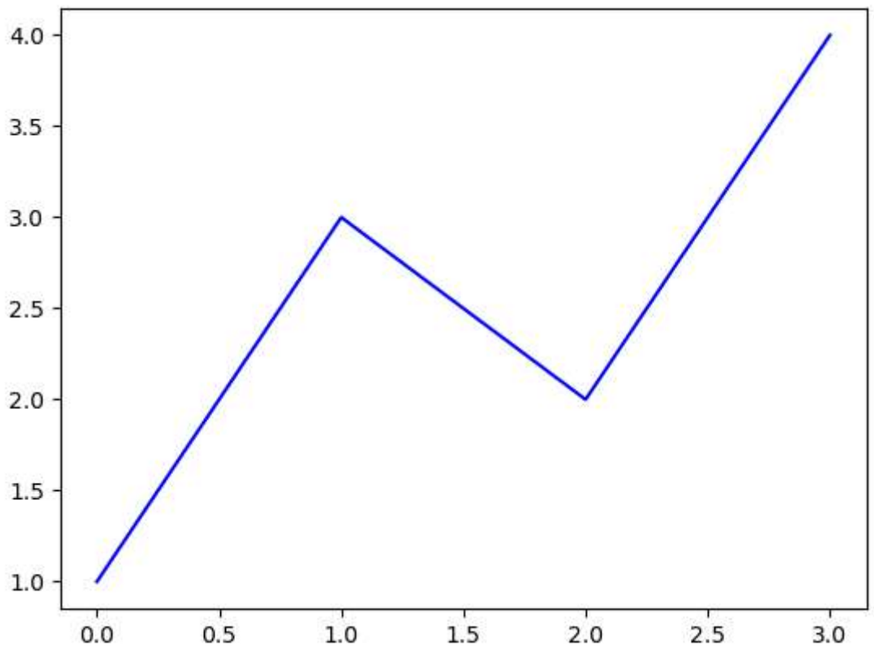
In [39]:
```
fig = plt.figure() # create a figure
ax1 = fig.add_subplot(2, 2, 1) # create the first subplot
ax2 = fig.add_subplot(2, 2, 2) # create the second subplot
ax3 = fig.add_subplot(2, 2, 3) # create the third subplot
ax4 = fig.add_subplot(2, 2, 4) # create the fourth subplot

plt.show() # show the plot
```



11. First plot with Matplotlib

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

In [ ]:
```
plt.plot([1, 3, 2, 4], 'b-') # plot the data with blue color and dashed line
plt.show( )
```



- 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).

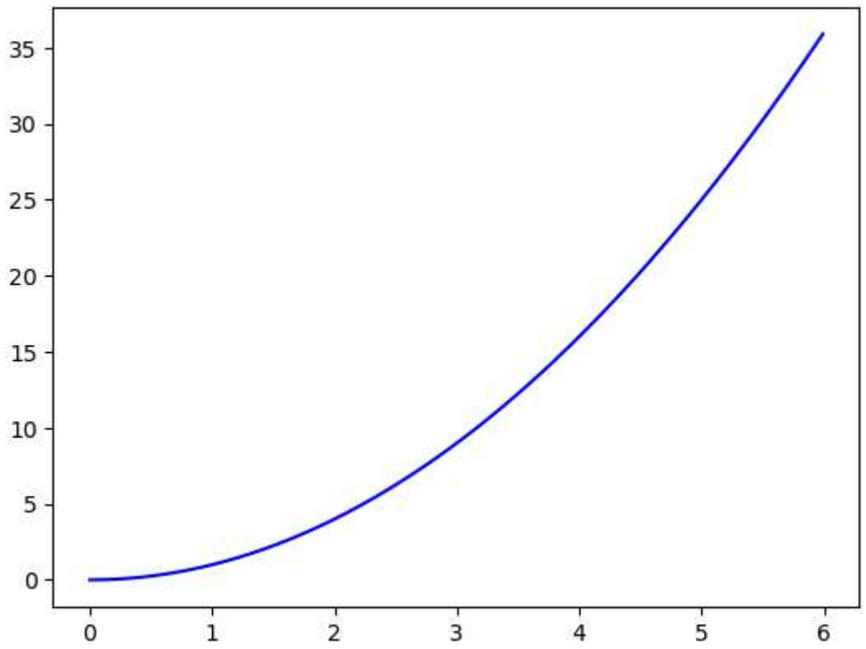
```
plt.plot([1, 3, 2, 4], 'b-')
```

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()
```

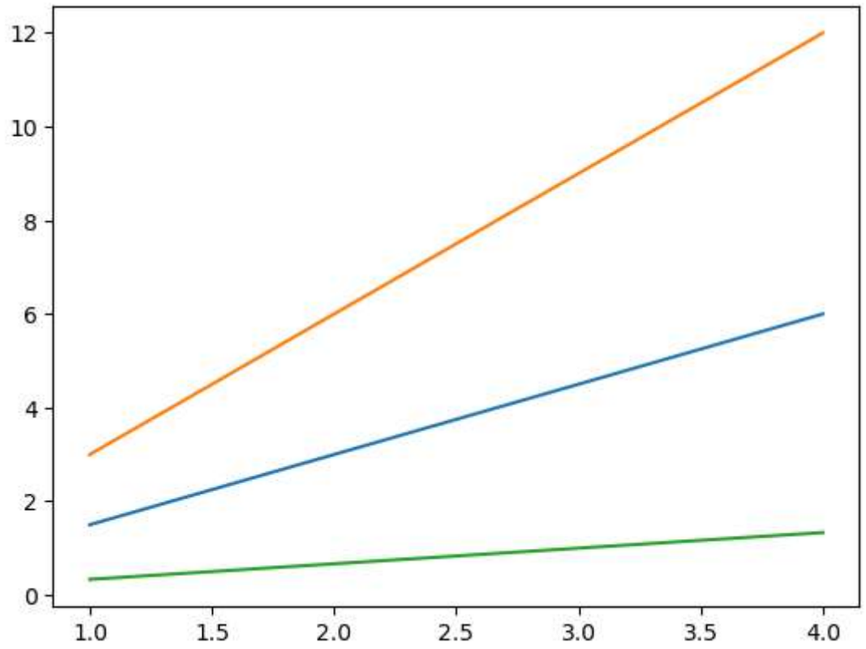
In [ ]:
```
x3 = np.arange(0.0, 6.0, 0.01) # create an array of 600 values between 0 and 6
plt.plot(x3, [xi**2 for xi in x3], 'b-') # plot the data with blue color and dashed line
plt.show() # show the plot
```



12. Multiline Plots

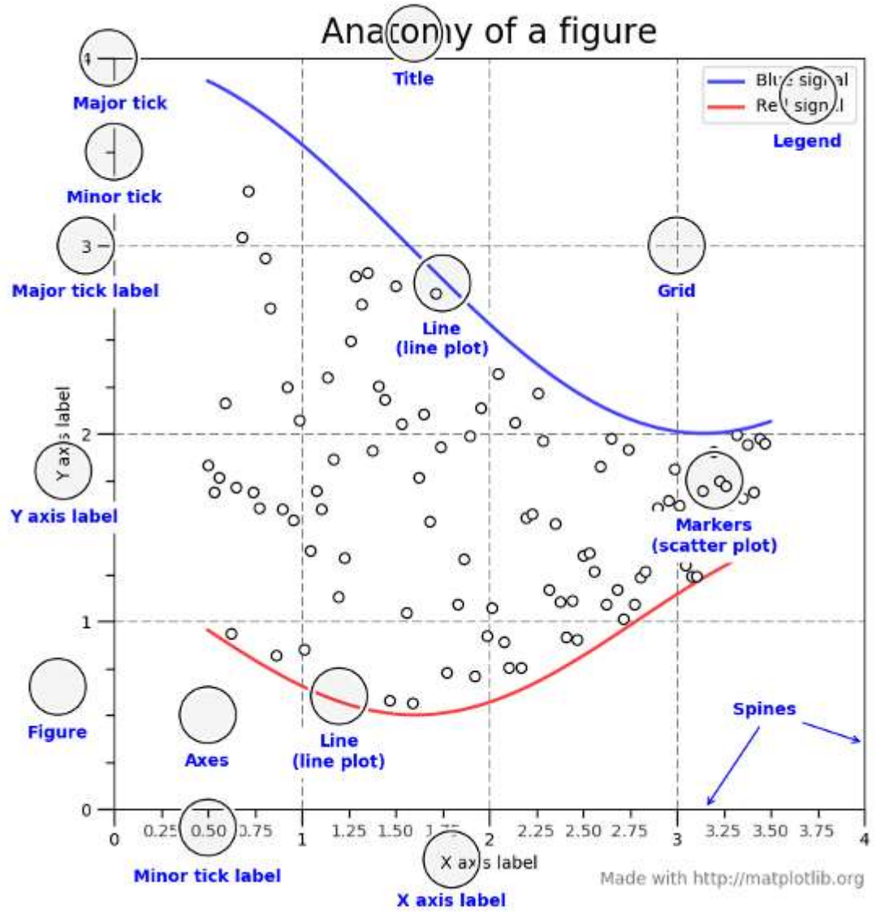
- Multiline Plots mean plotting more than one plot on the same figure. We can plot more than one plot on the same figure. It can be achieved by plotting all the lines before calling show(). It can be done as follows:-

In [ ]:
```
x4 = range(1, 5) # create an array of 4 values between 1 and 5
plt.plot(x4, [xi*1.5 for xi in x4]) # plot the data with blue color and dashed line
plt.plot(x4, [xi*3 for xi in x4]) # plot the data with blue color and dashed line
plt.plot(x4, [xi/3.0 for xi in x4]) # plot the data with blue color and dashed line
plt.show()
```



13. 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:-



14. Saving the Plot

We can save a figure in many different formats using the savefig() function.
Example:

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

To view the saved image inside Jupyter Notebook, we can use the IPython Image object:

```
from IPython.display import Image
Image('fig1.png')
```

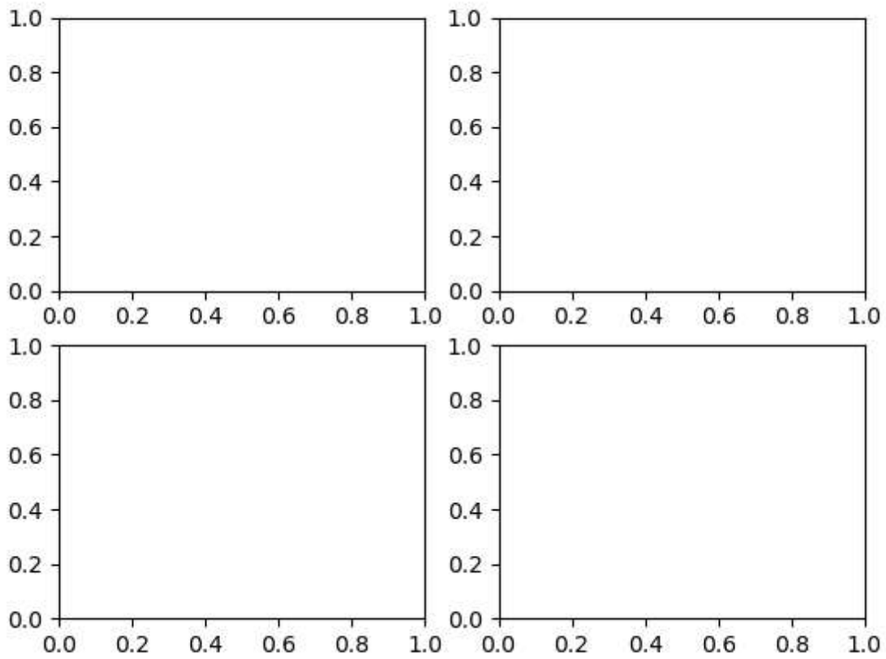
Note: The file format is automatically chosen based on the file extension (e.g., .png, .jpg, .pdf).
We can check all the supported file formats for saving by using:

```
fig.canvas.get_supported_filetypes()
```

In [ ]:

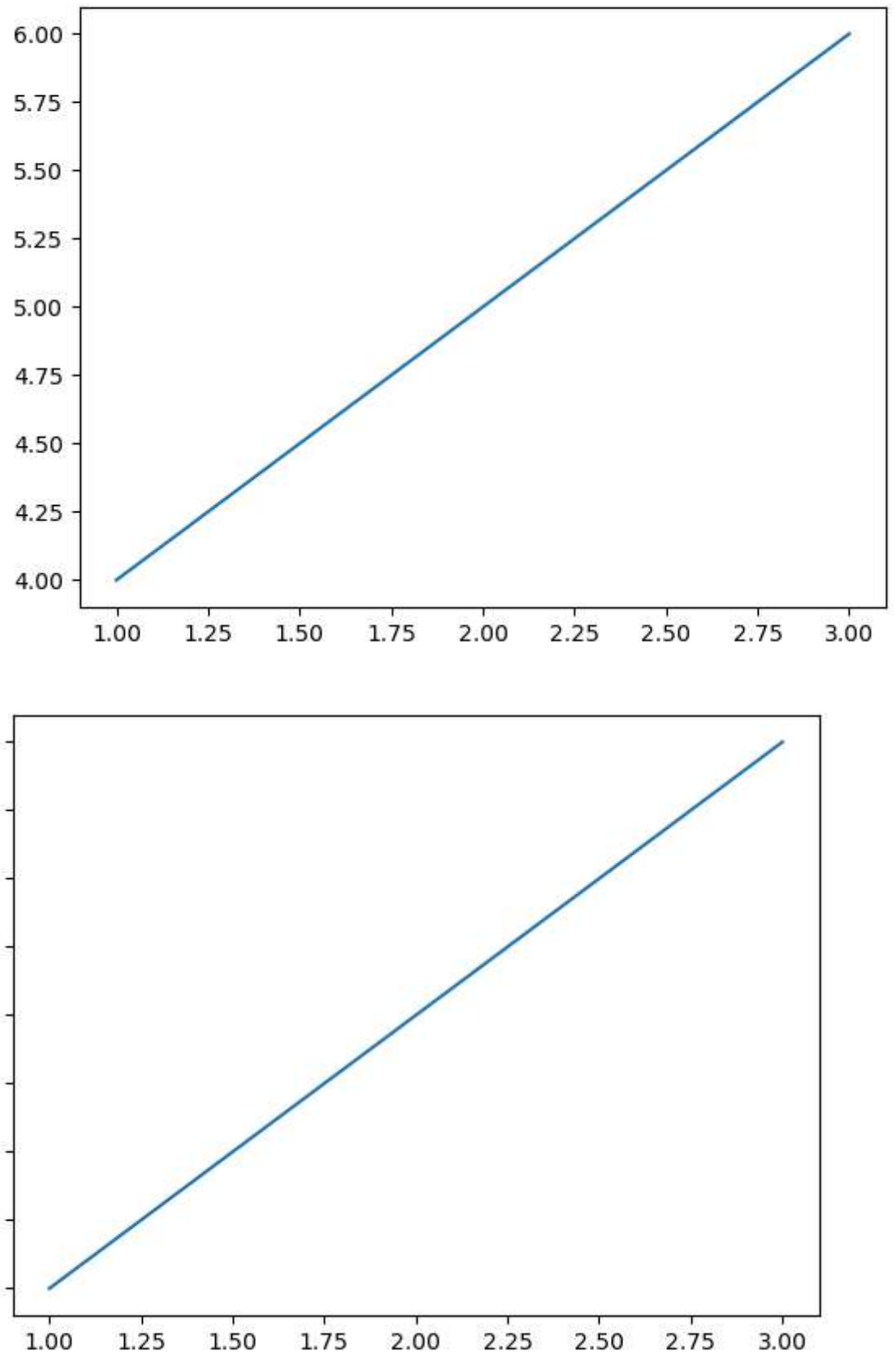
```
# Saving the figure
from IPython.display import Image # import the Image class from the IPython.display module
fig.savefig('plot2.png') # save the figure as a PNG file
# Explore the contents of figure
from IPython.display import Image # import the Image class from the IPython.display module
Image('plot2.png') # display the image
```

Out[ ]:



In [ ]:
```
# Step 1: Create figure
fig, ax = plt.subplots() # create a figure and an axes
ax.plot([1, 2, 3], [4, 5, 6]) # plot the data

# Step 2: Save figure
fig.savefig('plot1.png') # save the figure as a PNG file

# Step 3: Show saved figure inside notebook
from IPython.display import Image # import the Image class from the IPython.display module
Image('plot1.png') # display the image
```

Out[ ]:



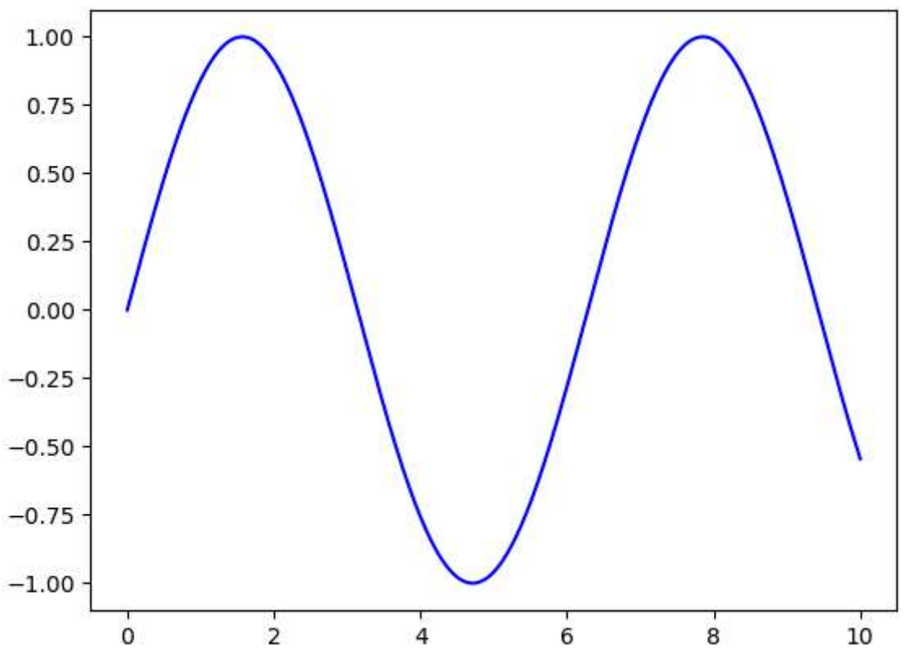
In [ ]:
```
# Explore supported file formats


fig.canvas.get_supported_filetypes() # get the supported file types for the current figure
```

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

15. Line Plot

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

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

ax = plt.axes() # create an axes

# Declare a variable x5
x5 = np.linspace(0, 10, 1000) # create an array of 1000 values between 0 and 10 (1000 points)


# Plot the sinusoid function
ax.plot(x5, np.sin(x5), 'b-'); # plot the sinusoid function with blue color and dashed line

plt.show()
```



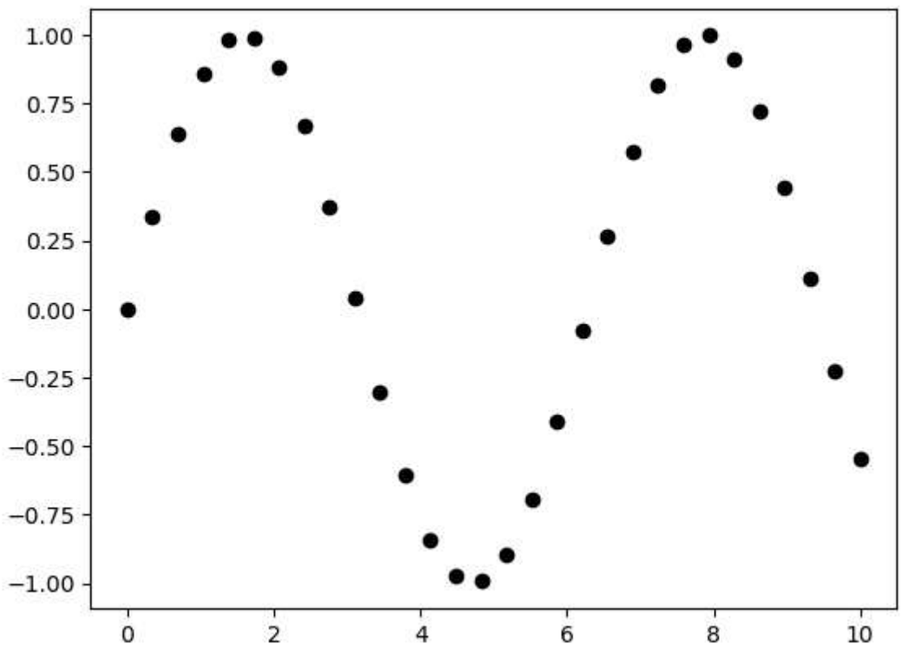
16. Scatter Plot

- A **scatter plot** is used to show individual data points, usually represented as **dots** or **circles**.
  It's useful for visualizing the relationship between two variables.

Scatter Plot with `plt.plot()`

We've already used `plt.plot()` or `ax.plot()` to create line plots.
The same functions can also create scatter plots by changing the **marker style**.

In [ ]:
```
x7 = np.linspace(0, 10, 30) # create an array of 30 values between 0 and 10 (30 points)

y7 = np.sin(x7) # create an array of 30 values between 0 and 10 (30 points)

plt.plot(x7, y7, 'o', color = 'black') # plot the data with black color and circles
plt.show() # show the plot
```

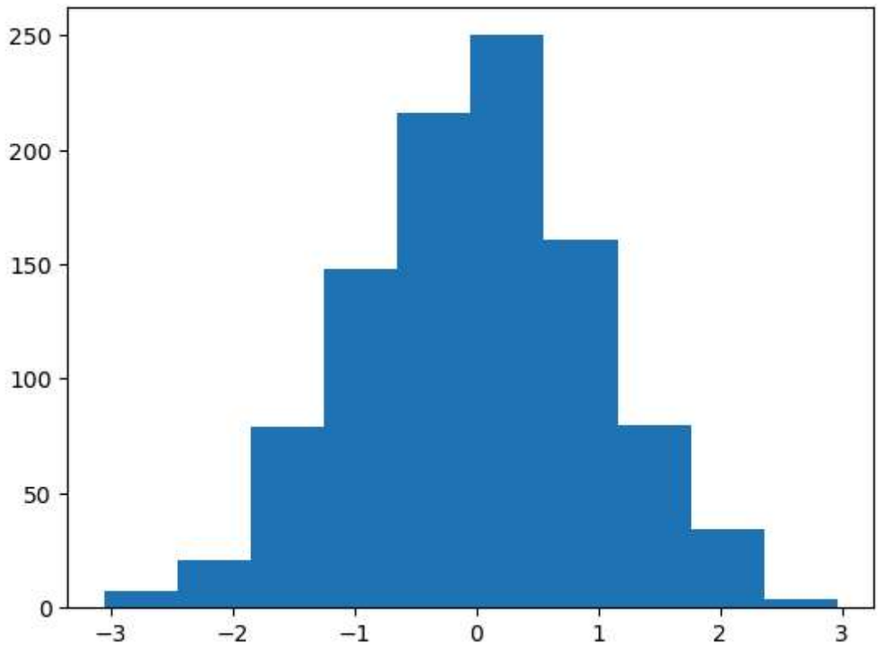


17. Histogram

A histogram is a graphical representation of the distribution of data. It uses bars to show the frequency of data values within specific intervals, called bins. Unlike a bar chart, a histogram groups values into continuous ranges rather than discrete categories.

We can create a histogram using **plt.hist():**

In [41]:
```
data1 = np.random.randn(1000)  # create an array of 1000 random numbers
plt.hist(data1); # plot the histogram of the data
plt.show() # show the plot
```

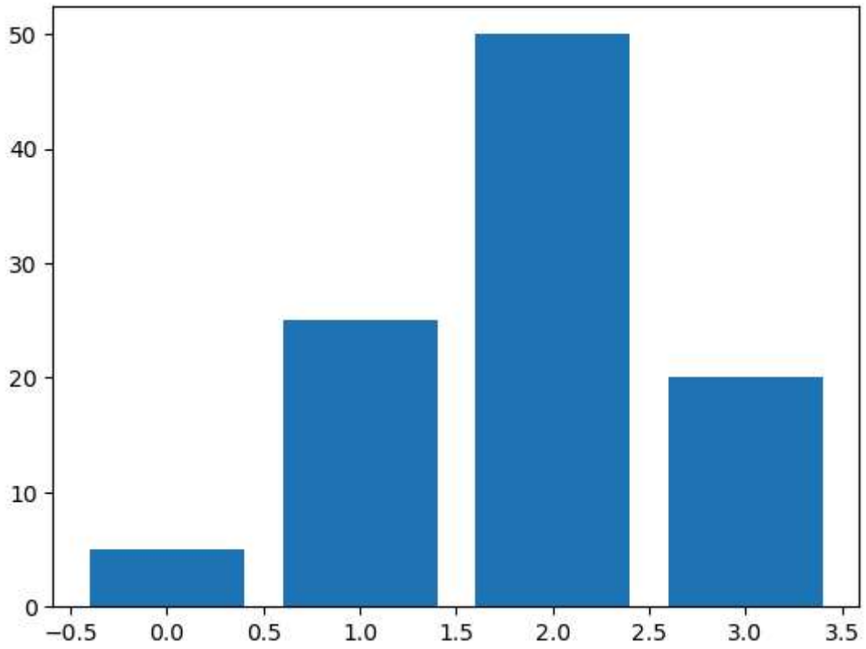


18. Bar Chart

A bar chart displays rectangular bars — either vertical or horizontal. The length of each bar is proportional to the value it represents. Bar charts are mainly used to compare two or more values.

We can create a bar chart using the **plt.bar()** function:

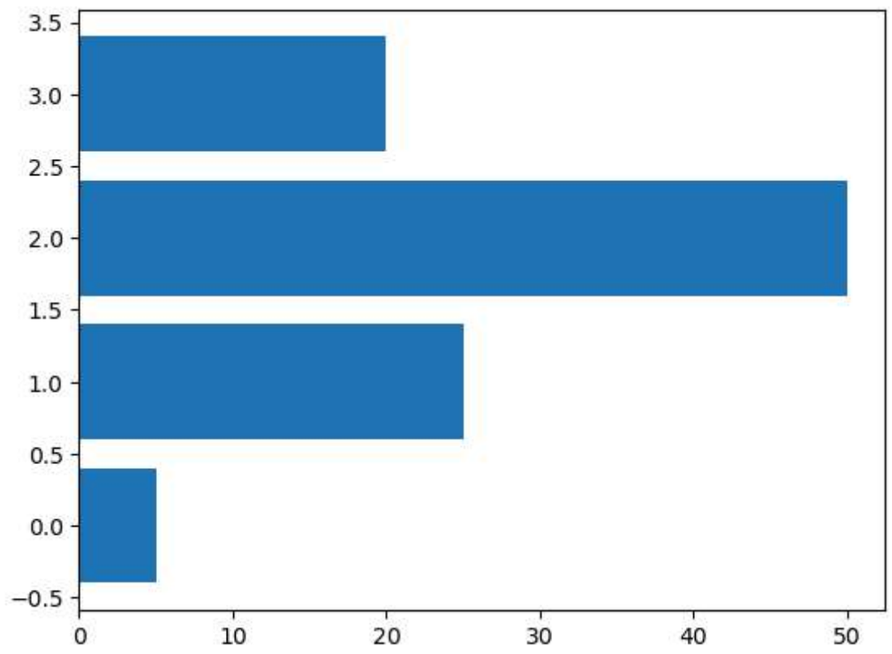
In [42]:
```
data2 = [5. , 25. , 50. , 20.] # create an array of 4 values between 5 and 20 (5, 25, 50, 20)
plt.bar(range(len(data2)), data2) # plot the data with bars
plt.show() # show the plot
```



19. Horizontal Bar Chart

A horizontal bar chart is like a regular bar chart, but bars are drawn horizontally. It is created using the **plt.barh()** function, which is the horizontal equivalent of **plt.bar()**.

In [43]:
```
data3 = [5. , 25. , 50. , 20.] # create an array of 4 values between 5 and 20 (5, 25, 50, 20)
plt.barh(range(len(data3)), data3) # plot the data with bars
plt.show() # show the plot
```



20. Error Bar Chart

In real-world experiments, measurements are **not always perfectly precise**.
To improve accuracy, we often repeat the measurements multiple times.

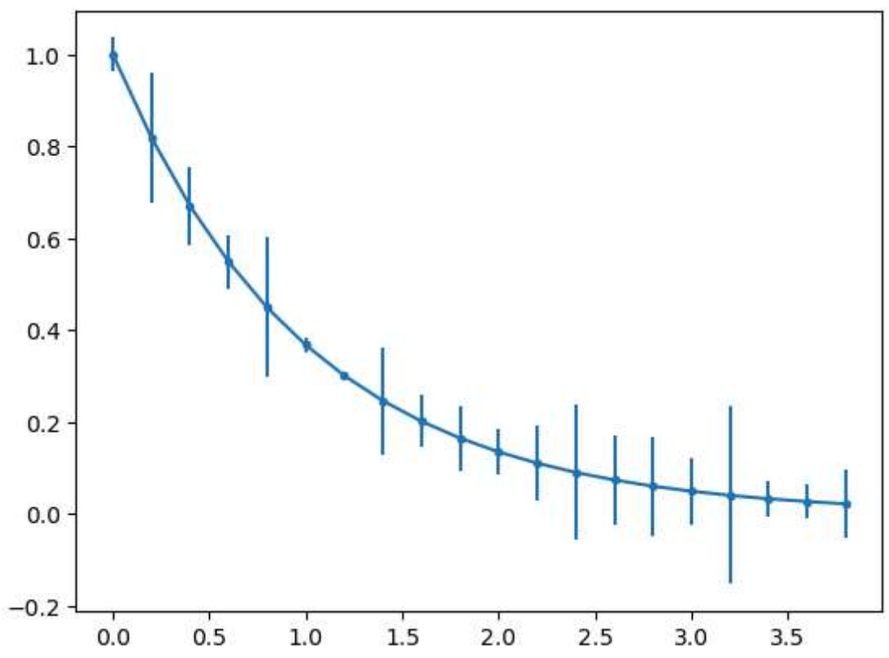
From these repeated measurements, we:

- Calculate the **mean value** (average).
- Show the **variability** (spread of data) using an **error bar**.

An **error bar chart** plots:

- A single data point (mean).
- Vertical or horizontal bars showing the range or standard deviation. In **Matplotlib**, we use the `errorbar()` function to create this chart.

In [ ]:
```
x9 = np.arange(0, 4, 0.2) # create an array of 20 values between 0 and 4 (20 points)
y9 = np.exp(-x9) # create an array of 20 values between 0 and 4 (20 points)
e1 = 0.1 * np.abs(np.random.randn(len(y9))) # create an array of 20 values between 0 and 4 (20 points)
plt.errorbar(x9, y9, yerr = e1, fmt = '.-') # plot the data with error bars
plt.show();
```



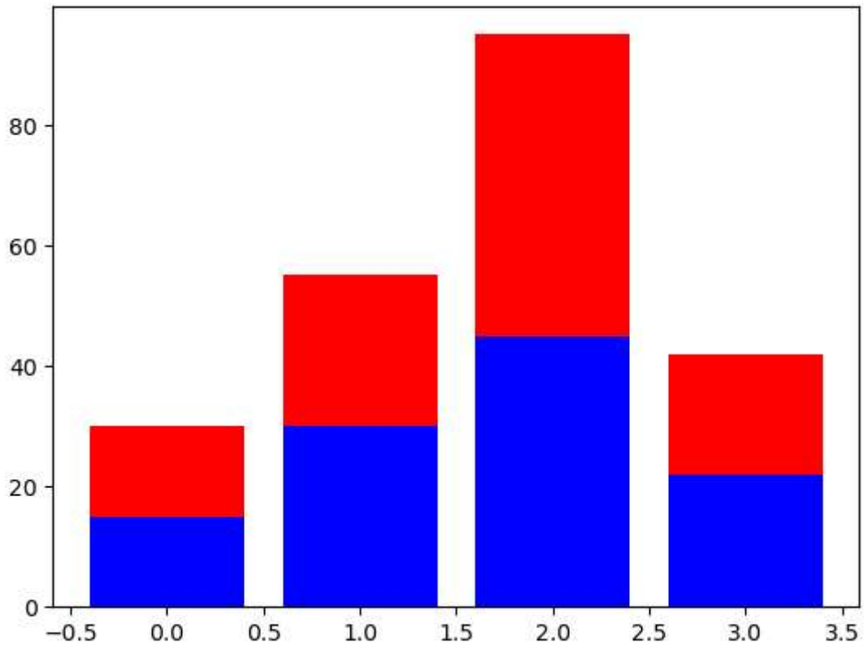
21. Stacked Bar Chart

A **stacked bar chart** is used to show different data series stacked on top of each other in the same bar.

In Matplotlib, we can create it using the `bottom` parameter of the `plt.bar()` function.
The `bottom` parameter tells Matplotlib **where to start** the next bar on top of the previous one.

In [44]:
```
A = [15., 30., 45., 22.]   # create an array of 4 values between 15 and 45 (15, 30, 45, 22)
B = [15., 25., 50., 20.]   # create an array of 4 values between 15 and 50 (15, 25, 50, 20)
```

```
z2 = range(4) # create an array of 4 values between 0 and 3 (0, 1, 2, 3)
plt.bar(z2, A, color = 'b') # plot the data with blue color and bars
plt.bar(z2, B, color = 'r', bottom = A) # plot the data with red color and bars
plt.show() # show the plot
```



In Matplotlib, the `bottom` parameter in `plt.bar()` lets us **set where a bar starts** instead of always starting from zero.

For example:

- The first `plt.bar()` call draws the **blue bars**.
- The second `plt.bar()` call draws the **red bars**, starting at the **top of the blue bars** using the `bottom` parameter.

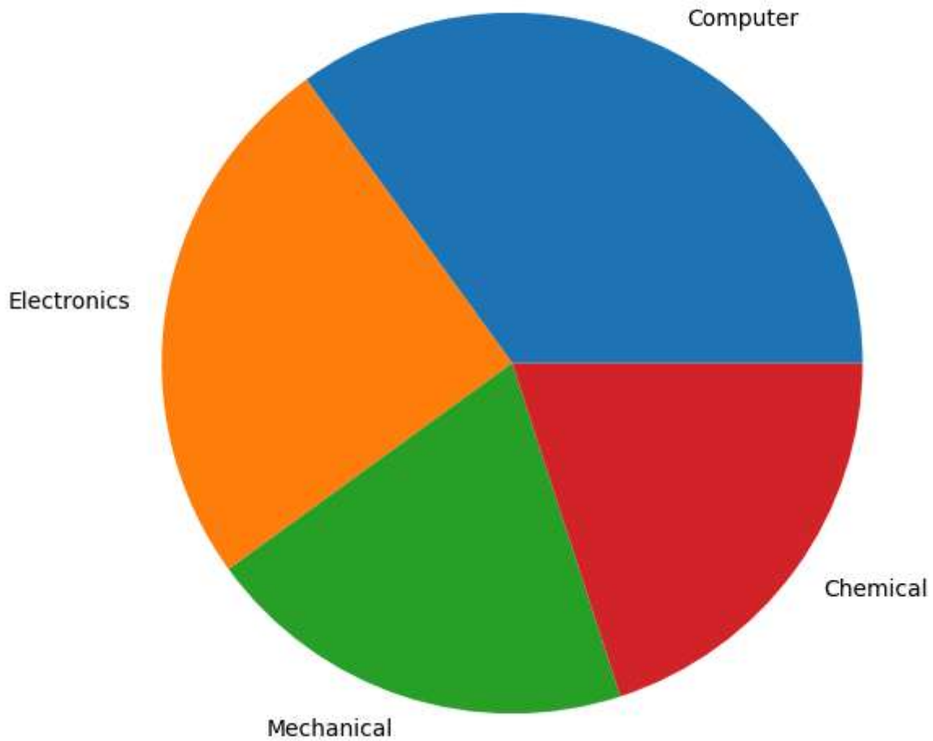
22. Pie Chart

A **pie chart** is a circular chart divided into slices, also called **wedges.**

- Each wedge shows a part of the whole.
- The size of each wedge is **proportional** to the value it represents.
- Pie charts are best when you want to **compare parts to the whole**, not necessarily parts to each other.

In **Matplotlib**, we can create a pie chart using the `pie()` function:

In [45]:
```
plt.figure(figsize=(7,7)) # create a figure with a size of 7x7
x10 = [35, 25, 20, 20] # create an array of 4 values between 20 and 35 (20, 25, 30, 35)
labels = ['Computer', 'Electronics', 'Mechanical', 'Chemical'] # create an array of 4 labels (Computer, Electronics, Mechanical, Chemical)
plt.pie(x10, labels=labels); # plot the data with labels
plt.show() # show the plot
```



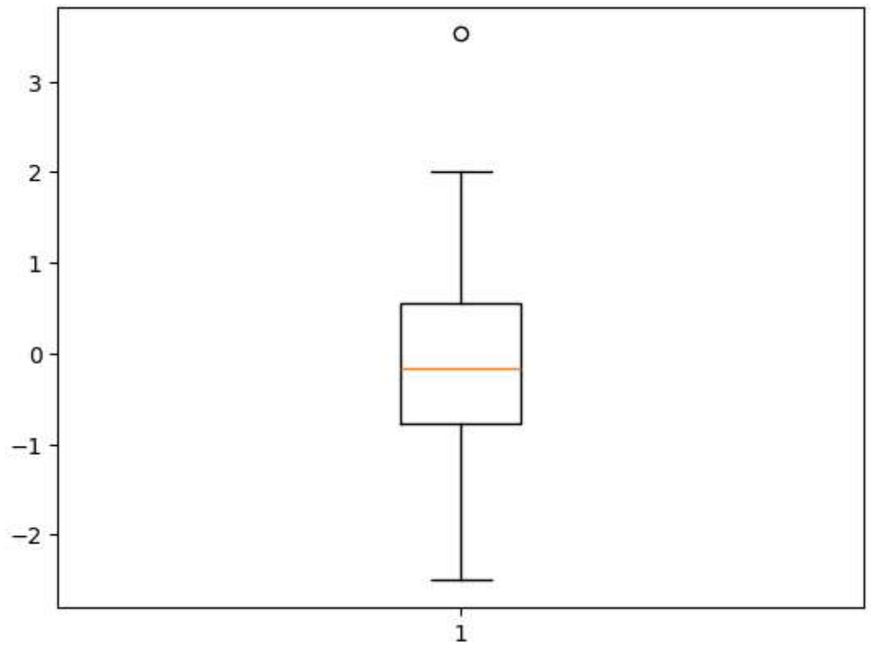
23. Boxplot

A **Boxplot** is a visual way to see the **distribution of data**.
It shows:

- **Median** (middle value)
- **Quartiles** (25% and 75% values)
- **Minimum and Maximum** values
- **Outliers** (if any)

We can create a boxplot in Matplotlib using the `boxplot()` function:

In [46]:
```
data3 = np.random.randn(100) # create an array of 100 random numbers
plt.boxplot(data3) # plot the data with a boxplot
plt.show() # show the plot
```



The `boxplot()` function helps visualize a dataset by showing key statistics like **median, quartiles, and outliers**.

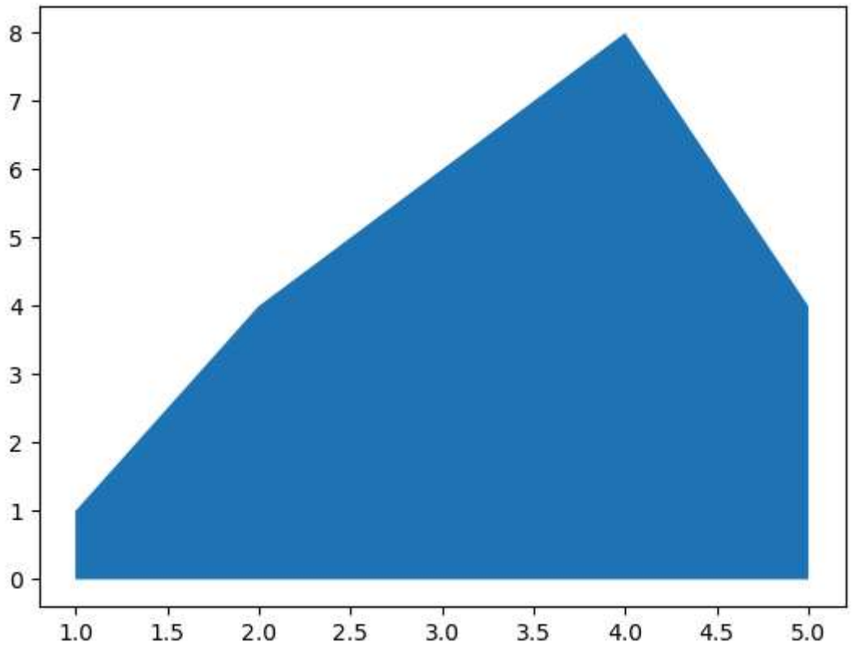
Here's what each part of a box plot represents:

- **Red Line (Median):** Shows the **middle value** of the data.
- **Blue Box (Interquartile Range, IQR):** Contains the **middle 50% of the data**, from the lower quartile (25th percentile) to the upper quartile (75th percentile). The box is centered around the median.
- **Lower Whisker:** Extends to the **smallest value** within 1.5 × IQR from the lower quartile.
- **Upper Whisker:** Extends to the **largest value** within 1.5 × IQR from the upper quartile.
- **Outliers:** Values beyond the whiskers are shown as **cross markers (×).**

24. Area Chart

An **Area Chart** is similar to a Line Chart, but the area **under the line is filled with color.**
It is useful to show how a value changes over time or another variable.

In [47]:
```
# Create some data
x12 = range(1, 6) # create an array of 5 values between 1 and 5 (1, 2, 3, 4, 5)
y12 = [1, 4, 6, 8, 4] # create an array of 5 values between 1 and 8 (1, 4, 6, 8, 4)
# Area plot
plt.fill_between(x12, y12) # plot the data with a filled area
plt.show()
```



- Area Chart in Matplotlib We can create a basic **Area Chart** in Matplotlib in two ways:
- Using `stackplot()`:

```
plt.stackplot(x12, y12)
```

```
Using fill_between() – this method is more flexible and easier to customize in the future:
```

25. Contour Plot

A **Contour Plot** helps us visualize **3D data in 2D** using lines or color-filled regions.
The lines on a contour plot are called **contour lines**, **level lines**, or **isolines.**
Each line represents points where a function has the **same value.**

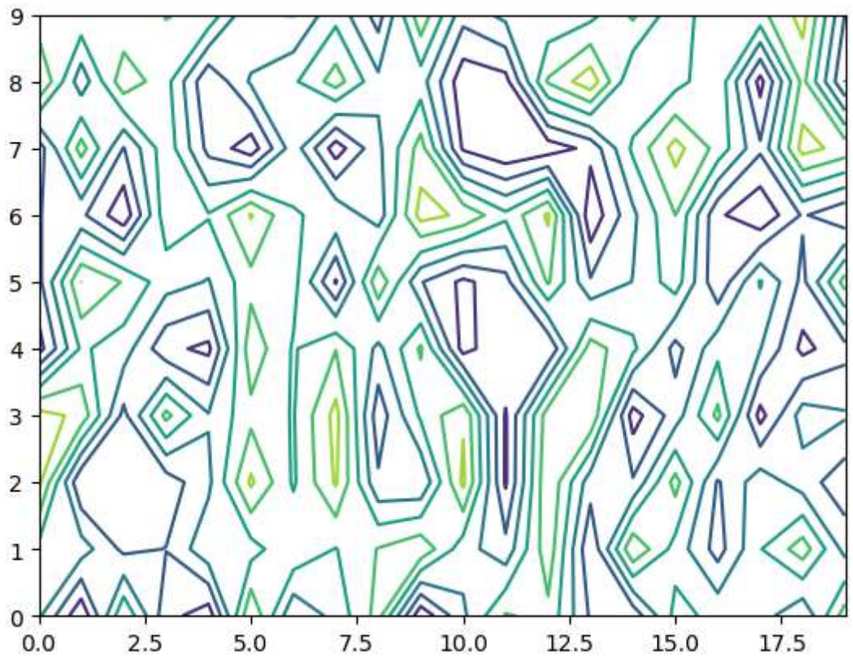
Key points about contour plots:

- The **density of lines** shows how steep the surface is.
  - Lines close together → steep slope
  - Lines far apart → gentle slope
- The **gradient** (direction of maximum change) is **always perpendicular** to contour lines.
- Contour lines start with "iso-" depending on what is measured, e.g., isotherms for temperature.

Applications:
Contour plots are widely used in:

- Meteorology → temperature, pressure, rainfall, wind speed
- Geography → terrain maps
- Engineering, magnetism, social sciences, and more

In [48]:
```
# Create a matrix
matrix1 = np.random.rand(10, 20) # create a matrix of 10x20 random numbers
cp = plt.contour(matrix1) # plot the matrix
plt.show() # show the plot
```



The `contour()` function is used to **draw contour lines** for a 2D dataset.

- It takes a **2D array (matrix)** as input.
- By default, Matplotlib chooses the number of contour lines automatically.
- You can also specify the number of lines using an extra parameter `N`.

26. Styles with Matplotlib Plots

Matplotlib comes with a **style module** (introduced in version 1.4) that makes it easy to change the **look and feel** of your plots.
You can use built-in styles or create your own custom styles.

In [49]:
```
# View list of all available styles

print(plt.style.available) # print the list of all available styles
```

```
['Solarize_Light2', '_classic_test_patch', '_mpl-gallery', '_mpl-gallery-nogrid', 'bmh', 'classic', 'dark_background', 'fast', 'fivethirtyeight', 'ggplot', 'grayscale', 'petroff10', '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:-

plt.style.use('seaborn-bright')

- Setting Styles for Plots

Matplotlib allows us to change the **style** of plots to make them look better.

For example, we can use the `seaborn-bright` style:

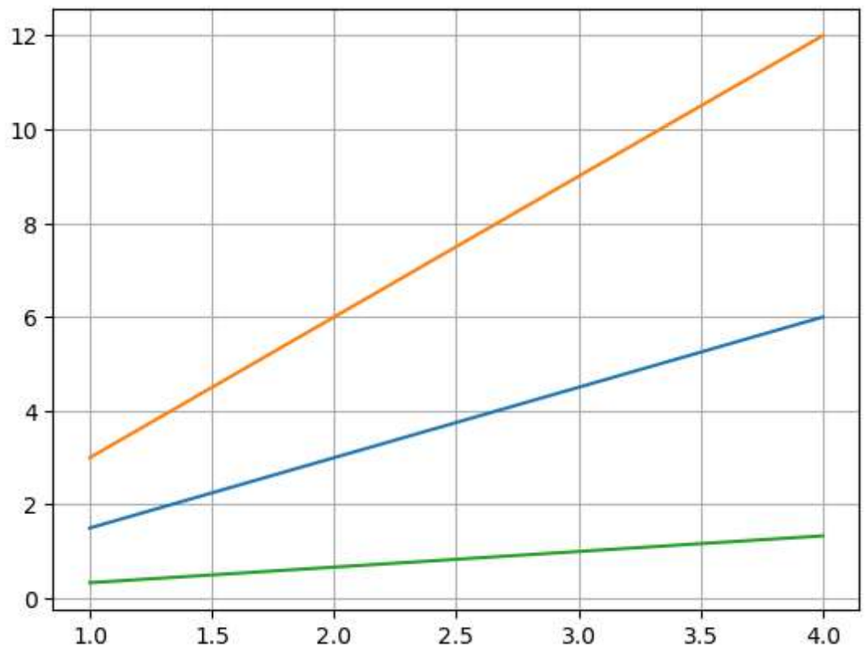
```
plt.style.use('seaborn-bright')
```

27. Adding a Grid

Sometimes a plot looks empty and it's hard to read the values.
Adding a **grid** makes it easier to see where points lie and improves understanding.

We can add a grid using the `grid()` function. It takes a **Boolean value**:

In [50]:
```
x15 = np.arange(1, 5) # create an array of 4 values between 1 and 5 (1, 2, 3, 4)
plt.plot(x15, x15*1.5, x15, x15*3.0, x15, x15/3.0) # plot the data with a grid
plt.grid(True) # show the grid
plt.show() # show the plot
```

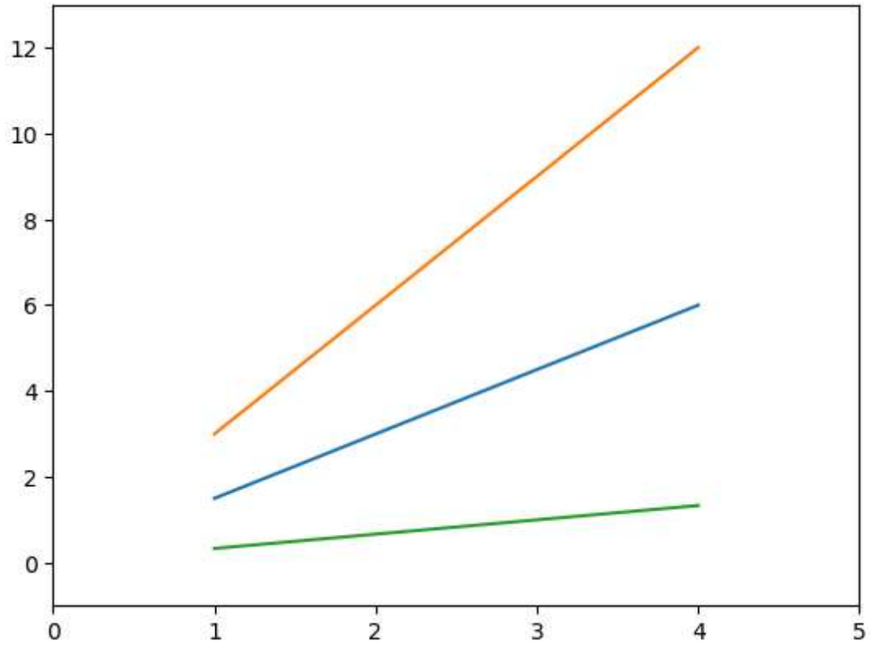


28. Handling Axes

By default, **Matplotlib** automatically chooses the limits of the x-axis and y-axis to fit the data.

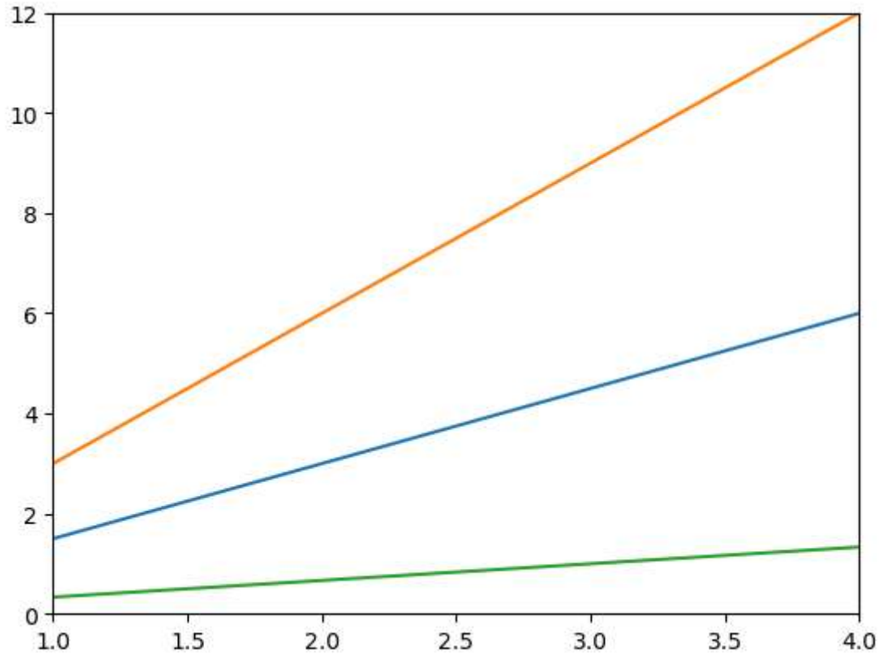
Sometimes, we may want to **set the axes limits ourselves**.
We can do this using the `axis()` function:

In [51]:
```
x15 = np.arange(1, 5) # create an array of 4 values between 1 and 5 (1, 2, 3, 4)
plt.plot(x15, x15*1.5, x15, x15*3.0, x15, x15/3.0) # plot the data with a grid
plt.axis()   # shows the current axis limits values
plt.axis([0, 5, -1, 13]) # set the axis limits
plt.show() # show the plot
```



- 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 [52]:
```
x15 = np.arange(1, 5) # create an array of 4 values between 1 and 5 (1, 2, 3, 4)
plt.plot(x15, x15*1.5, x15, x15*3.0, x15, x15/3.0) # plot the data with a grid
plt.xlim([1.0, 4.0]) # set the x-axis limits
plt.ylim([0.0, 12.0]) # set the y-axis limits
plt.show()
```



Controlling the Legend Location

In Matplotlib, the **legend** shows labels for different plot elements.
You can control where the legend appears using the `loc` argument in `ax.legend()`.

Common `loc` values:

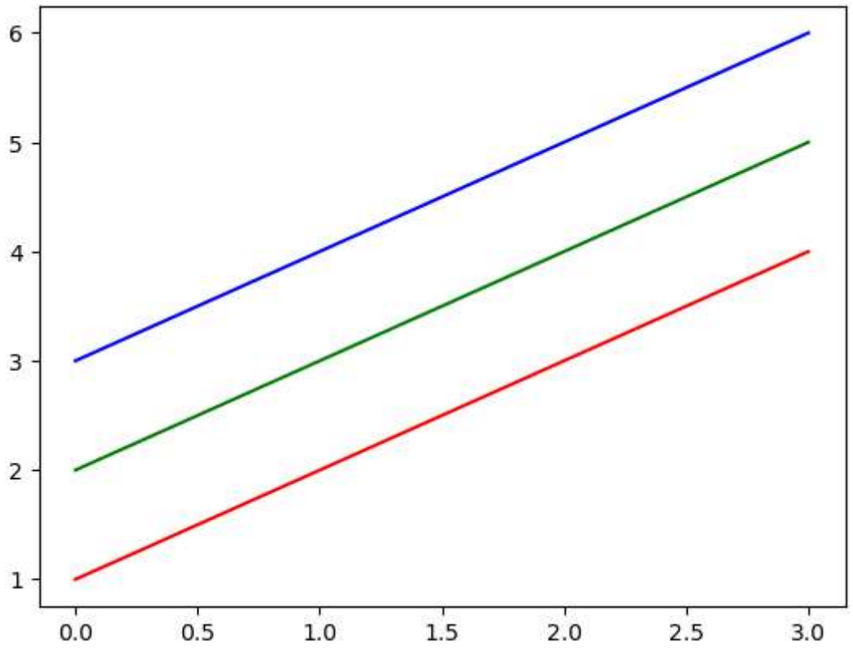
```
ax.legend(loc=0)  # Let Matplotlib choose the best location
ax.legend(loc=1)  # Upper right corner
ax.legend(loc=2)  # Upper left corner
ax.legend(loc=3)  # Lower left corner
ax.legend(loc=4)  # Lower right corner
ax.legend(loc=5)  # Right
ax.legend(loc=6)  # Center left
ax.legend(loc=7)  # Center right
ax.legend(loc=8)  # Lower center
ax.legend(loc=9)  # Upper center
ax.legend(loc=10) # Center
```

33. Controlling Colours

In Matplotlib, we can make lines or curves appear in **different colours** to make plots more readable and attractive.

For example, we can draw **red**, **blue**, and **green** lines by specifying the colour in the plot command:

In [53]:
```
x16 = np.arange(1, 5) # create an array of 4 values between 1 and 5 (1, 2, 3, 4)
plt.plot(x16, 'r') # plot the data with red color
plt.plot(x16+1, 'g') # plot the data with green color
plt.plot(x16+2, 'b') # plot the data with blue color
plt.show() # show the plot
```



Colors in Matplotlib

Matplotlib allows you to set colors in plots in different ways.
The easiest way is by using **color abbreviations**:

Abbreviation	Color Name
b	blue
c	cyan
g	green
k	black
m	magenta
r	red
w	white
y	yellow

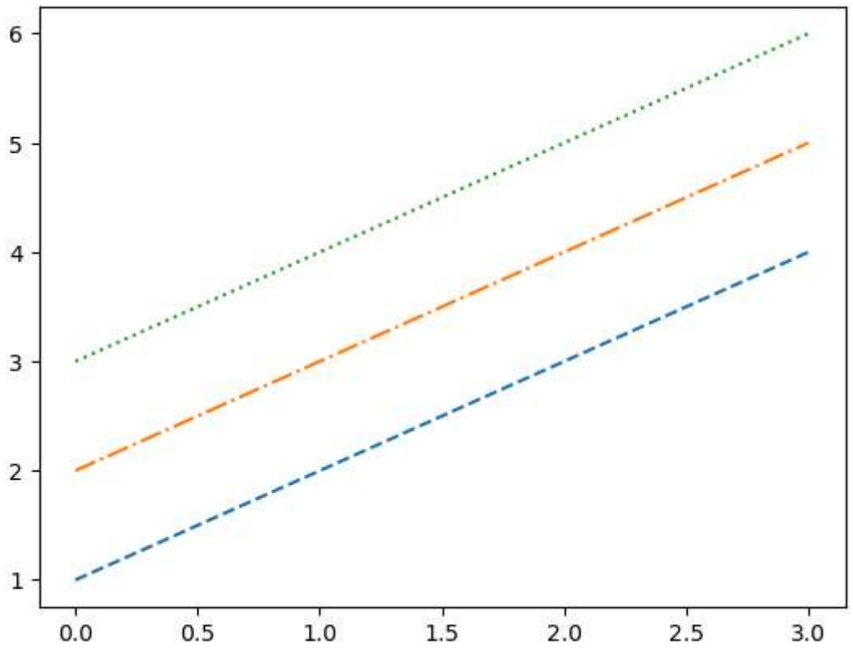
- The full colour name, such as yellow
- Hexadecimal string such as ##FF00FF
- RGB tuples, for example (1, 0, 1)
- Grayscale intensity, in string format such as '0.7'.

34. Controlling Line Styles

In Matplotlib, you can change the **style of lines** in your plots to make them look different.

For example, you can use **solid**, **dashed**, **dotted**, or **dash-dot** lines to distinguish multiple plots.

In [54]:
```
x16 = np.arange(1, 5) # create an array of 4 values between 1 and 5 (1, 2, 3, 4)
plt.plot(x16, '--', x16+1, '-.', x16+2, ':') # plot the data with different styles
plt.show() # show the plot
```



Line Styles in Matplotlib

We can change how a line looks using **line styles**.

For example, the following code creates:

- A **blue dashed line**
- A **green dash-dot line**
- A **red dotted line**

All the line styles in Matplotlib are represented by these abbreviations:

Style Abbreviation	Style Description
`-`	Solid line
`--`	Dashed line
`-.`	Dash-dot line
`:`	Dotted line

Tip: The default format string for a single line plot is `'b-'`, which means a **blue solid line**.

35. Summary

In this project, we explored **Matplotlib**, Python's basic plotting library, and learned how to create and customize different types of charts.

Key points covered:

1. **Matplotlib Basics**
   - Object hierarchy and architecture
   - **Pyplot API** and **Object-Oriented API**
   - Using **subplots** to create multiple plots in one figure
2. **Types of Plots**
   - Line Plot
   - Scatter Plot
   - Histogram
   - Bar Chart & Horizontal Bar Chart
   - Pie Chart
   - Box Plot
   - Area Chart
   - Contour Plot
3. **Customization Techniques**
   - Applying **styles** to plots
   - Adding **grid lines**
   - Handling **axes** and **ticks**
   - Adding **labels, titles, and legends**
   - Customizing **colors** and **line styles**