

Matplotlib

Matplotlib is a Python library that is specifically designed to do effective data visualization. Matplotlib is an open-source Python library that offers various data visualization (like Line plots, histograms, scatter plots, bar charts, Scatter plots, Pie Charts, and Area Plot etc). Generally, matplotlib overlays two APIs:

- **The pyplot API:** To make plot using matplotlib.pyplot.
- **Object-Oriented API:** A group of objects assembled with greater flexibility than pyplot. It provides direct access to Matplotlib's backend layers.

Why To Learn Matplotlib?

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It has become one of the most widely used plotting libraries in the Python ecosystem. Some of the reasons are as to make Matplotlib popular:

- **Plotting Capabilities:** Matplotlib provides extensive functionality for creating a variety of plots like line plots, scatter plots, bar charts, histograms, pie charts, 3D plots, etc.
- **Quality Graphics:** It allows its users to control every aspect of their plots, including colours, line styles, markers, fonts, and annotations.
- **Integration with NumPy and Pandas:** Matplotlib works with NumPy and Pandas to visualize arrays, data frames, and other data structures.
- **Cross-Platform Compatibility:** Matplotlib operates on Windows, macOS, and Linux, making it accessible to many people.
- **Extensive Documentation and Tutorials:** Beginners and experts may easily get started with Matplotlib thanks to its extensive documentation and online training.

Matplotlib is a robust and versatile Python toolkit used for visualizing data which makes it indispensable for data analysts, scientists, engineers, and other professionals working with data.

Key Features

Simple Plotting: Matplotlib allows us to create basic plots easily with just a few lines of code.

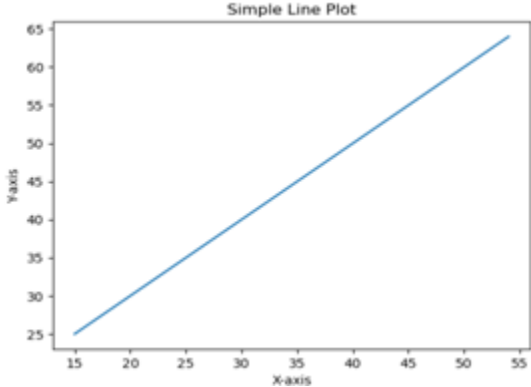
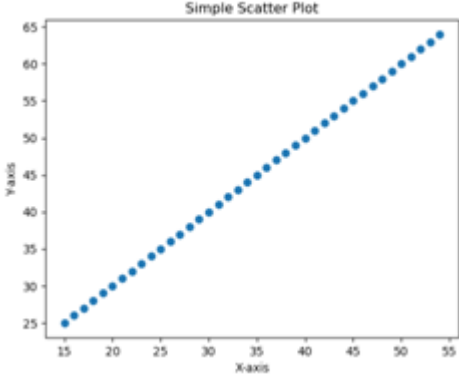
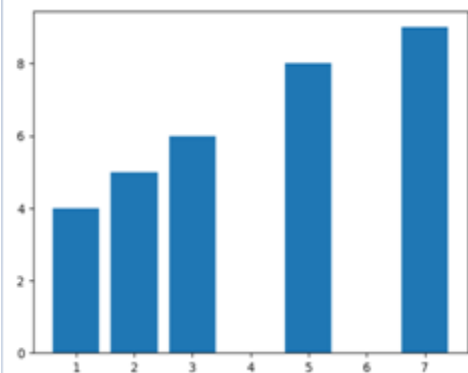
Customization: We can extensively customize plots by adjusting colors, line styles, markers, labels, titles and more.

Multiple Plot Types: It supports a wide variety of plot types such as line plots, scatter plots, bar charts, histograms, pie charts, 3D plots, etc.

Publication Quality: Matplotlib produces high-quality plots suitable for publications and presentations with customizable DPI settings.

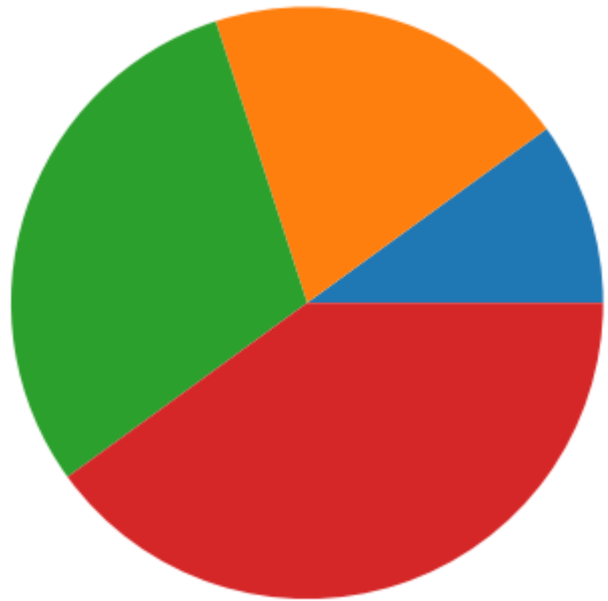
Support for LaTeX Typesetting: We can use LaTeX for formatting text and mathematical expressions in plots.

Types of plot

Name of the plot	Definition	Image
Line plot	<p>A line plot is a type of graph that displays data points connected by straight line segments.</p> <p>The <code>plt.plot()</code> function of the matplotlib library is used to create the line plot.</p>	 <p>A line plot titled 'Simple Line Plot' showing a linear relationship. The x-axis is labeled 'X-axis' and ranges from 15 to 55. The y-axis is labeled 'Y-axis' and ranges from 25 to 65. A single blue line connects the points (15, 25) and (55, 65).</p>
Scatter plot	<p>A scatter plot is a type of graph that represents individual data points by displaying them as markers on a two-dimensional plane.</p> <p>The <code>plt.scatter()</code> function is used to plot the scatter plot.</p>	 <p>A scatter plot titled 'Simple Scatter Plot' showing a linear relationship. The x-axis is labeled 'X-axis' and ranges from 15 to 55. The y-axis is labeled 'Y-axis' and ranges from 25 to 65. Numerous blue dots represent individual data points, forming a straight line from (15, 25) to (55, 65).</p>
Bar plot	<p>A bar plot or bar chart is a visual representation of categorical data using rectangular bars.</p> <p>The <code>plt.bar()</code> function is used to plot the bar plot.</p>	 <p>A bar plot showing categorical data. The x-axis has labels 1, 2, 3, 4, 5, 6, and 7. The y-axis ranges from 0 to 8. The bars have heights of approximately 4, 5, 6, 0, 8, 0, and 9 respectively.</p>

Pie plot

A pie plot is also known as a pie chart. It is a circular statistical graphic used to illustrate numerical proportions. It divides a circle into sectors or slices to represent the relative sizes or percentages of categories within a dataset. The `plt.pie()` function is used to plot the pie chart.



The above mentioned are the basic plots of the matplotlib library. We can also visualize the 3-d plots with the help of Matplotlib.

Note: We can create multiple plots within a single figure using subplots. This is useful when we want to display multiple plots together.

Details

matplotlib.pyplot is a collection of command style functions that make Matplotlib work like MATLAB. Each Pyplot function makes some change to a figure. For example, a function creates a figure, a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.

Types of Plots

Sr.No	Function & Description
1	Bar Make a bar plot.
2	Barh Make a horizontal bar plot.
3	Boxplot Make a box and whisker plot.
4	Hist Plot a histogram.
5	hist2d Make a 2D histogram plot.
6	Pie Plot a pie chart.
7	Plot Plot lines and/or markers to the Axes.
8	Polar Make a polar plot..
9	Scatter Make a scatter plot of x vs y.
10	Stackplot Draws a stacked area plot.
11	Stem Create a stem plot.
12	Step Make a step plot.
13	Quiver Plot a 2-D field of arrows.

Image Functions

Sr.No	Function & Description
1	Imread Read an image from a file into an array.
2	Imsave Save an array as in image file.
3	Imshow Display an image on the axes.

Axis Functions

Sr.No	Function & Description
1	Axes Add axes to the figure.
2	Text Add text to the axes.
3	Title Set a title of the current axes.
4	Xlabel Set the x axis label of the current axis.
5	Xlim Get or set the x limits of the current axes.
6	Xscale .
7	Xticks Get or set the x-limits of the current tick locations and labels.
8	Ylabel Set the y axis label of the current axis.
9	Ylim Get or set the y-limits of the current axes.
10	Yscale Set the scaling of the y-axis.

11	Yticks Get or set the y-limits of the current tick locations and labels.
----	--

Figure Functions

Sr.No	Function & Description
1	Figtext Add text to figure.
2	Figure Creates a new figure.
3	Show Display a figure.
4	Savefig Save the current figure.
5	Close Close a figure window.

Creating a Simple Plot Using Matplotlib

The following are the key steps to create a simple plot –

Step 1: Preparing the Data

It involves in preparing the data to be plotted. This could be numeric values, lists, arrays or data loaded from files.

Step 2: Plotting Function

Using Matplotlib's pyplot API or object-oriented interface a simple plot is generated. The most common functions include the below functions.

- **plot()**: Creates line plots.
- **scatter()**: Generates scatter plots.
- **bar()**: Plots bar charts.
- **hist()**: Creates histograms. And more

Step 3: Customization (Optional)

Customization involves adding labels, titles, adjusting colors, markers, axes limits, legends and other attributes to enhance the plot's readability and aesthetics.

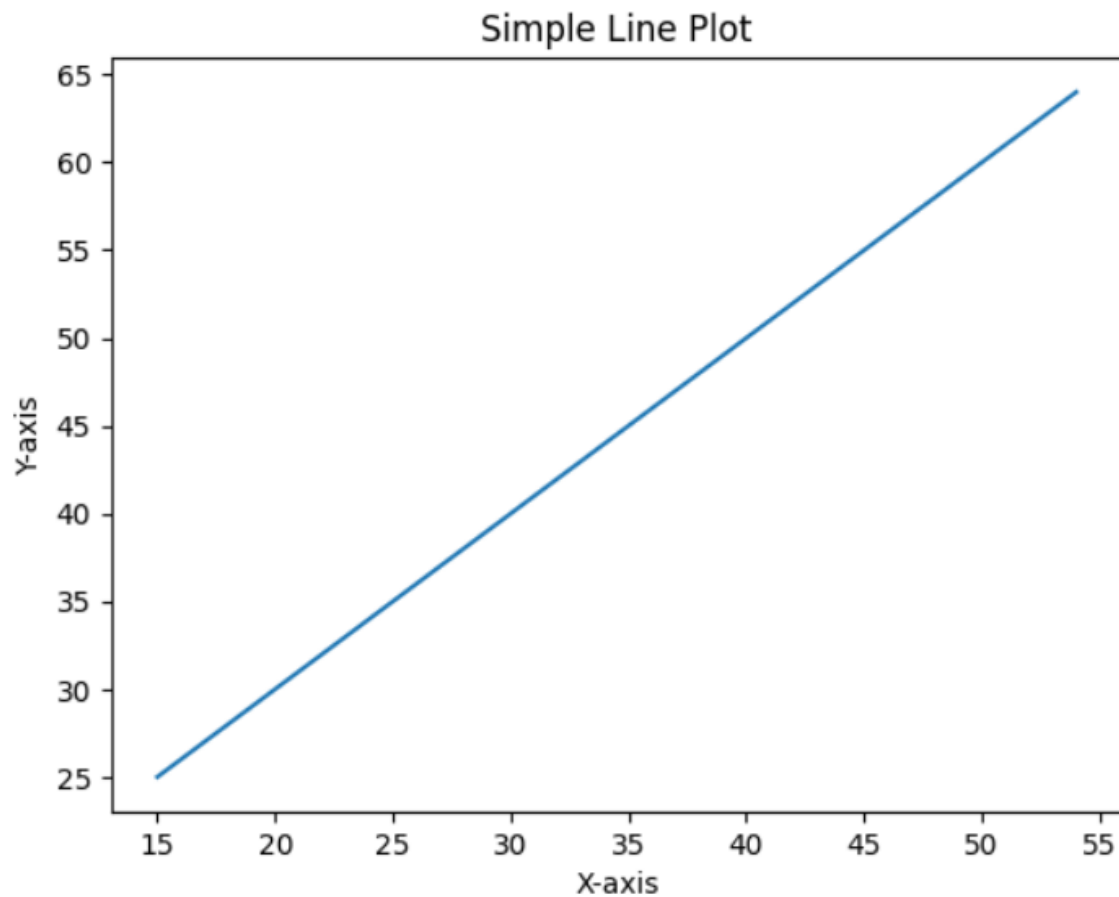
Step 4: Displaying the Plot

Finally displaying the plot using `show()` in pyplot or other appropriate methods.

```
import matplotlib.pyplot as plt
# Data
x = [(i-5) for i in range(20,60)]
y = [(i+5) for i in range(20,60)]
plt.plot(x,y)

# Customize the plot (optional)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Line Plot')

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

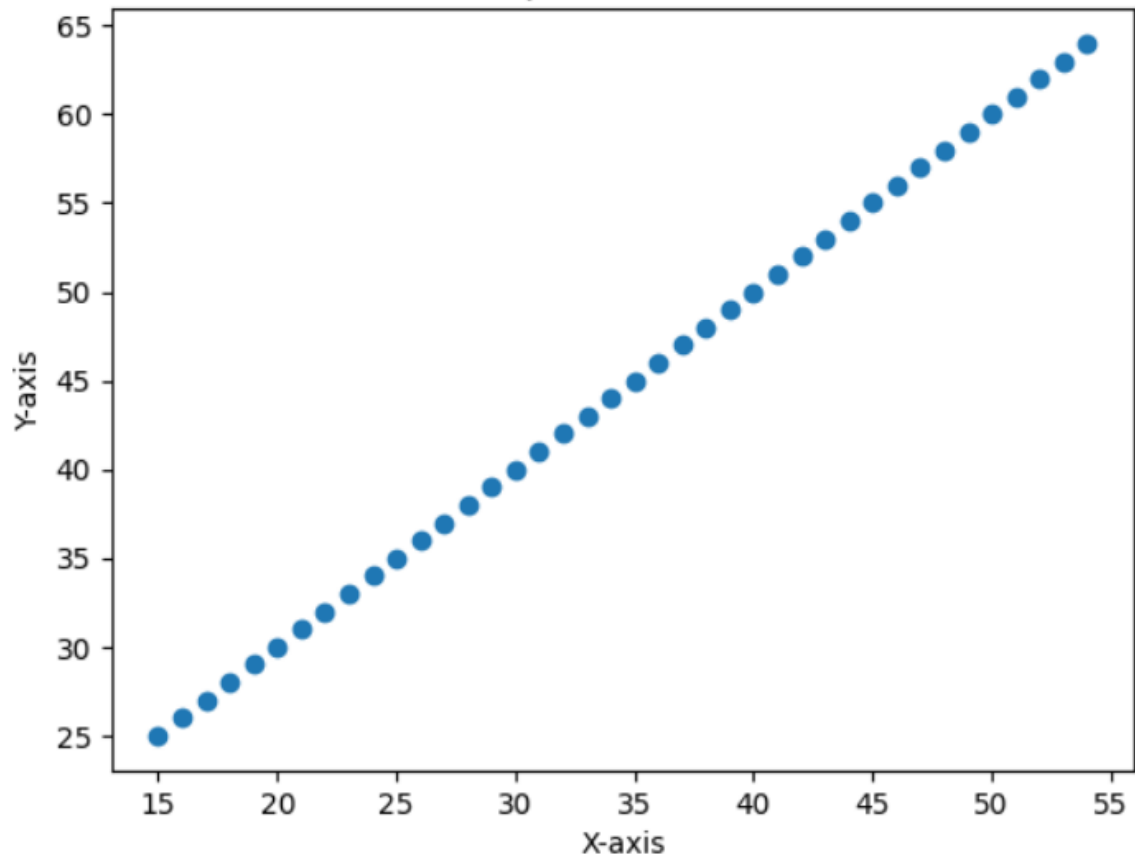


```
import matplotlib.pyplot as plt
# Data
x = [(i-5) for i in range(20,60)]
y = [(i+5) for i in range(20,60)]
plt.scatter(x,y)

# Customize the plot (optional)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Scatter Plot')

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


Simple Scatter Plot



Saving the Figures

Matplotlib library provides the `savefig()` function for to save the plot that we have created.

Common File Formats for Saving

PNG (.png): Good for general-purpose images which supports transparency.

JPEG (.jpg): Suitable for images with smooth gradients but may lose some quality due to compression.

PDF (.pdf): Ideal for vector-based images scalable without loss of quality.

SVG (.svg): Scalable Vector Graphics which suitable for web-based or vector-based graphics.

Syntax

```
plt.savefig(fname, dpi=None, bbox_inches='tight', pad_inches=0.1, format=None, kwargs)
```

Where,

fname: The file name or path of the file to save the figure. The file extension determines the file format such as ".png", ".pdf".

dpi: Dots per inch i.e. resolution for the saved figure. Default is "None" which uses the Matplotlib default.

bbox_inches: Specifies which part of the figure to save. Options include 'tight', 'standard' or a specified bounding box in inches.

pad_inches: Padding around the figure when `bbox_inches='tight'`.

format: Explicitly specify the file format. If 'None' the format is inferred from the file extension in `fname`.

kwargs: Additional keyword arguments specific to the chosen file format.

Markers






In Matplotlib markers are used to highlight individual data points on a plot. The marker parameter in the plot() function is used to specify the marker style. The following is the syntax for using markers in Matplotlib.



```
plt.plot(x, y, marker='marker_style')
```

Where,

x and y: Arrays or sequences of values representing the data points to be plotted.

Marker: Specifies the marker style to be used. It can be a string or one of the following marker styles:

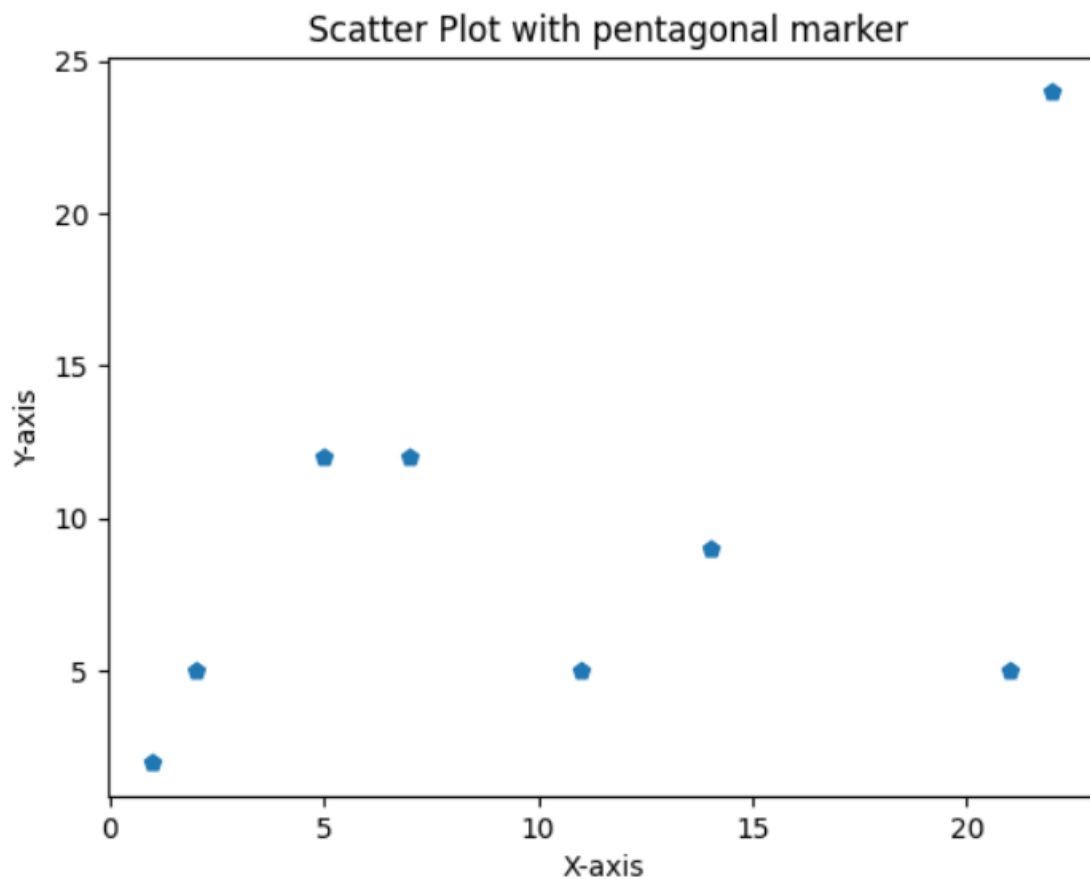
Sr.No.	Marker & Definition
1	 Point marker
2	 Pixel marker
3	 Circle marker
4	 Triangle down marker
5	 Triangle up marker
6	Triangle left marker
7	 Triangle right marker
8	 Downward-pointing triangle marker
9	 Upward-pointing triangle marker
10	 Left-pointing triangle marker
11	 Right-pointing triangle marker

12	 Square marker
13	 Pentagon marker
14	 Star marker
15	 Hexagon marker (1)
16	 Hexagon marker (2)
17	 Plus marker
18	 Cross marker
19	 Diamond marker
20	 Thin diamond marker
21	 Horizontal line marker

```
import matplotlib.pyplot as plt
# Data
x = [22,1,7,2,21,11,14,5]
y = [24,2,12,5,5,5,9,12]
plt.scatter(x,y, marker = 'p')

# Customize the plot (optional)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title(' Scatter Plot with pentagonal marker')

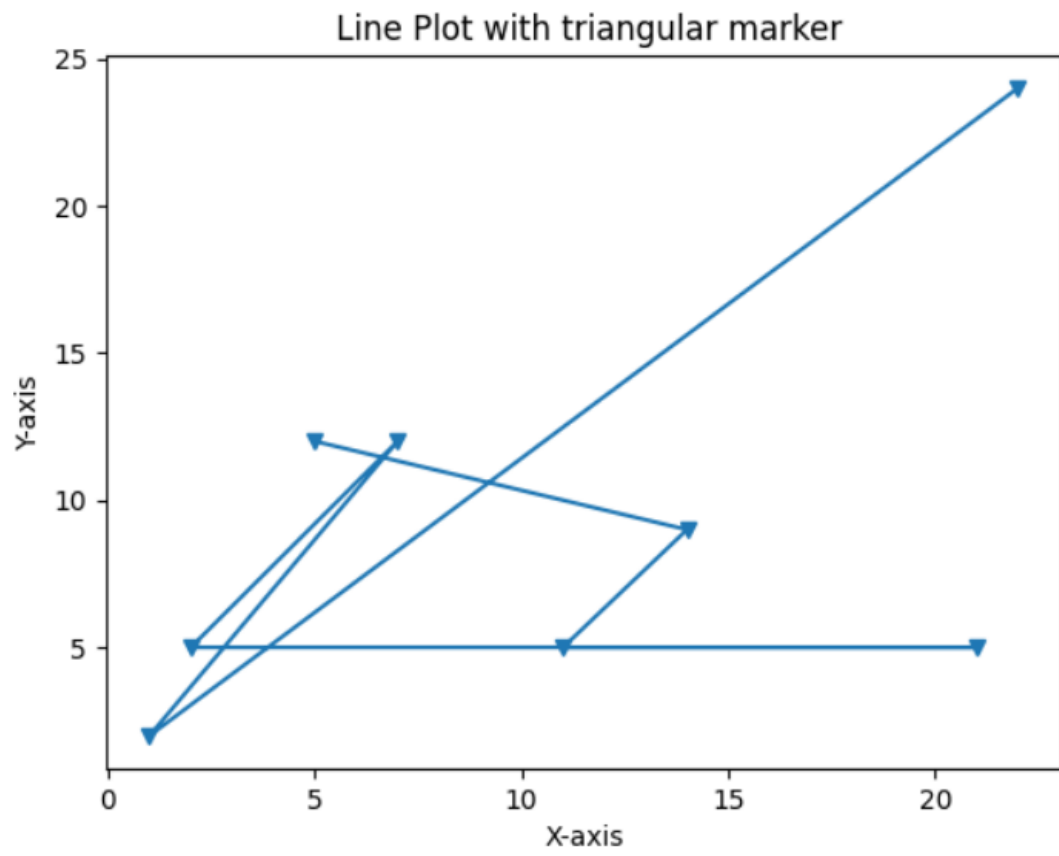
# Display the plot
plt.show()
```



```
import matplotlib.pyplot as plt
# Data
x = [22,1,7,2,21,11,14,5]
y = [24,2,12,5,5,5,9,12]
plt.plot(x,y, marker = 'v')

# Customize the plot (optional)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title(' Line Plot with triangular marker')

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



Figures

In Matplotlib library a figure is the top-level container that holds all elements of a plot or visualization. It can be thought of as the window or canvas where plots are created. A single figure can contain multiple subplots i.e. axes, titles, labels, legends and other elements. The `figure()` function in Matplotlib is used to create a new figure and the syntax used is:

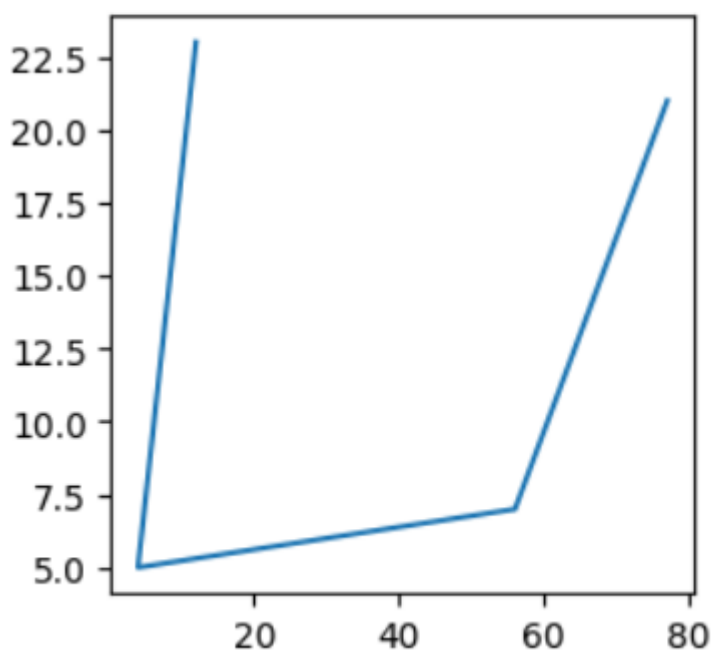
```
plt.figure(figsize=(width, height), dpi=resolution)
```

Where,

figsize = (width, height): Specifies the width and height of the figure in inches. This parameter is optional.

Dpi = resolution: Sets the resolution or dots per inch for the figure. Optional and by default is 100.

```
import matplotlib.pyplot as plt
# Create a figure
plt.figure(figsize=(3, 3), dpi=100)
x = [12,4,56,77]
y = [23,5,7,21]
plt.plot(x,y)
plt.show()
```



Customizing Figures

To display and customize the figures we have the functions **plt.show()**, **plt.title()**, **plt.xlabel()**, **plt.ylabel()**, **plt.legend()**.

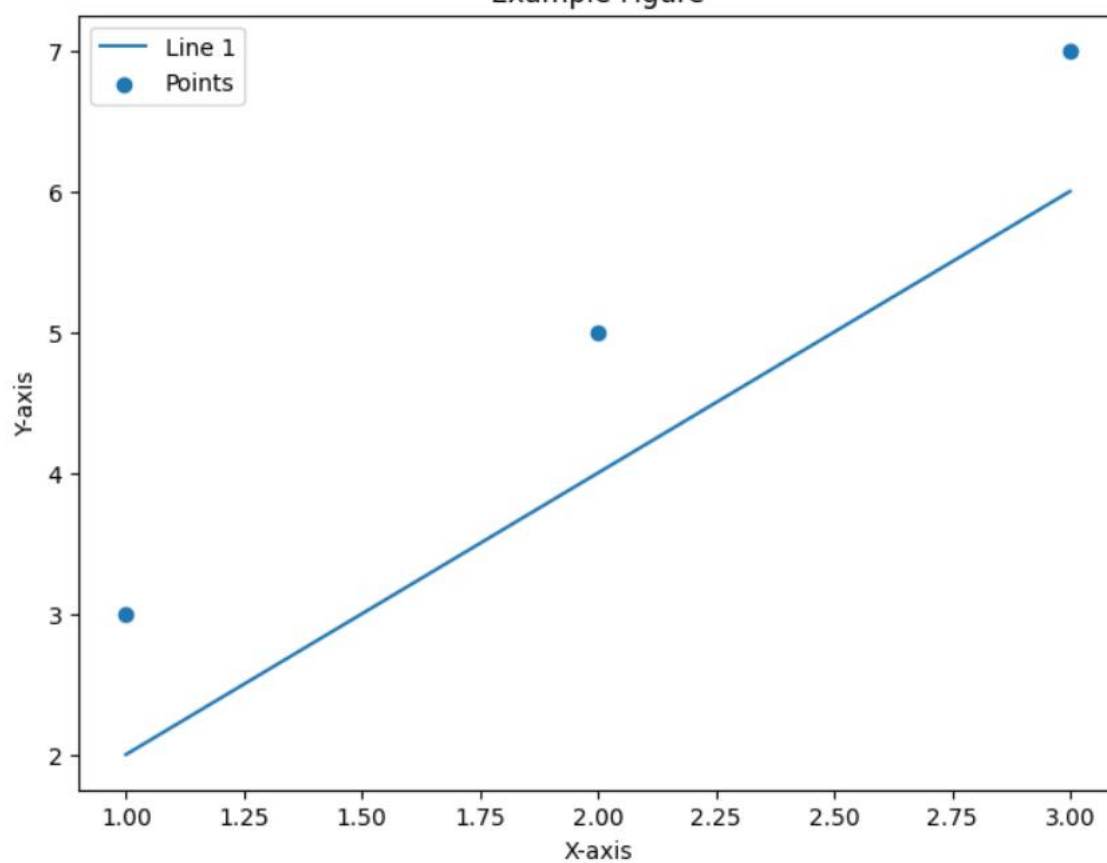
```
import matplotlib.pyplot as plt
# Create a figure
plt.figure(figsize=(8, 6))

# Add plots or subplots within the figure
plt.plot([1, 2, 3], [2, 4, 6], label='Line 1')
plt.scatter([1, 2, 3], [3, 5, 7], label='Points')

# Customize the figure
plt.title('Example Figure')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.legend()

# Display the figure
plt.show()
```


Example Figure



Styles

Matplotlib comes with a variety of built-in styles that offer different color schemes, line styles, font sizes and other visual properties.

Examples include ggplot, seaborn, classic, dark_background and more.

```
import matplotlib.pyplot as plt
plt.style.use('ggplot') # Setting the 'ggplot' style
```

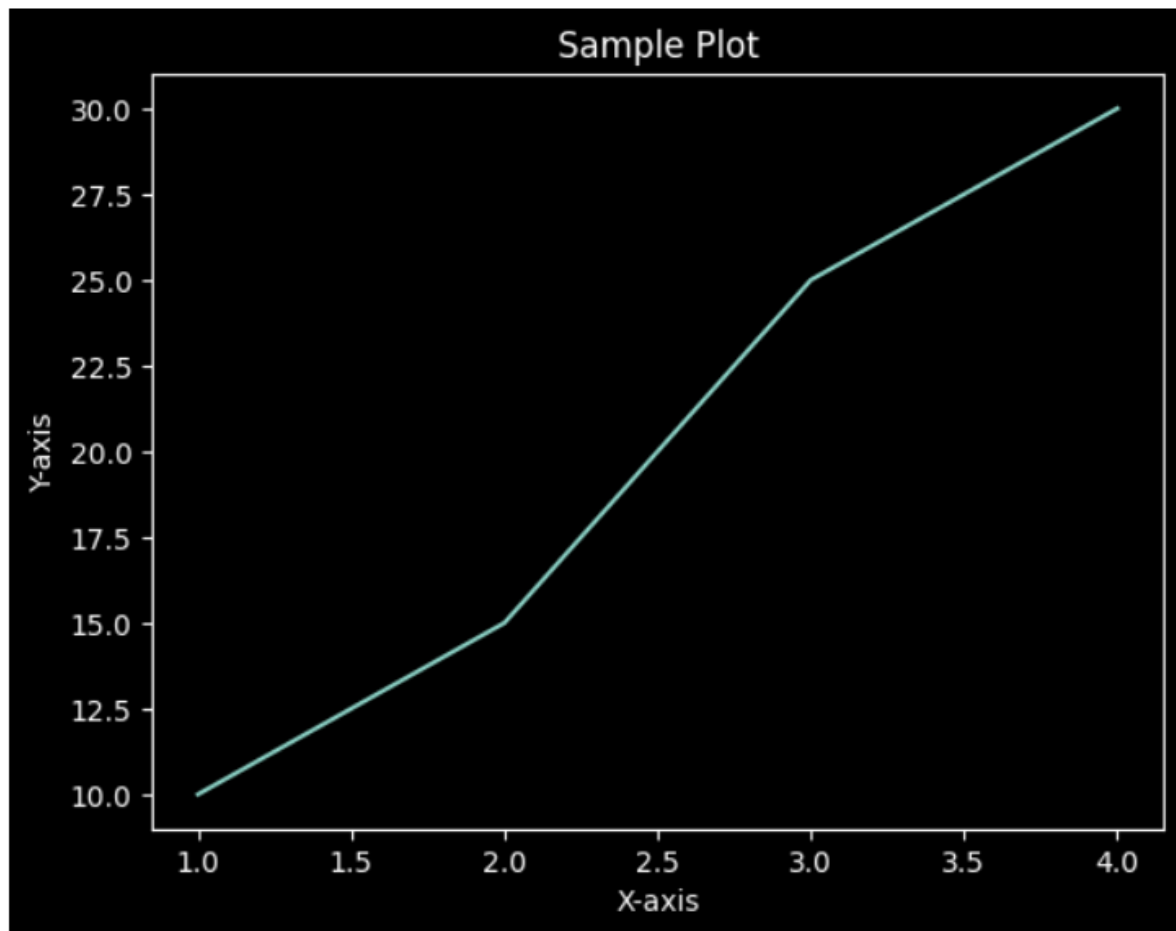
```
import matplotlib.pyplot as plt
print(plt.style.available) # Prints available styles
```

```
['Solarize_Light2', '_classic_test_patch', '_mpl-gallery', '_mpl-gallery-nogrid',
```

We can create custom style files with specific configurations and then use `plt.style.use('path_to_custom_style_file')` to apply them.

```
import matplotlib.pyplot as plt
# Using a specific style
plt.style.use('dark_background')

# Creating a sample plot
plt.plot([1, 2, 3, 4], [10, 15, 25, 30])
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Sample Plot')
plt.show()
```

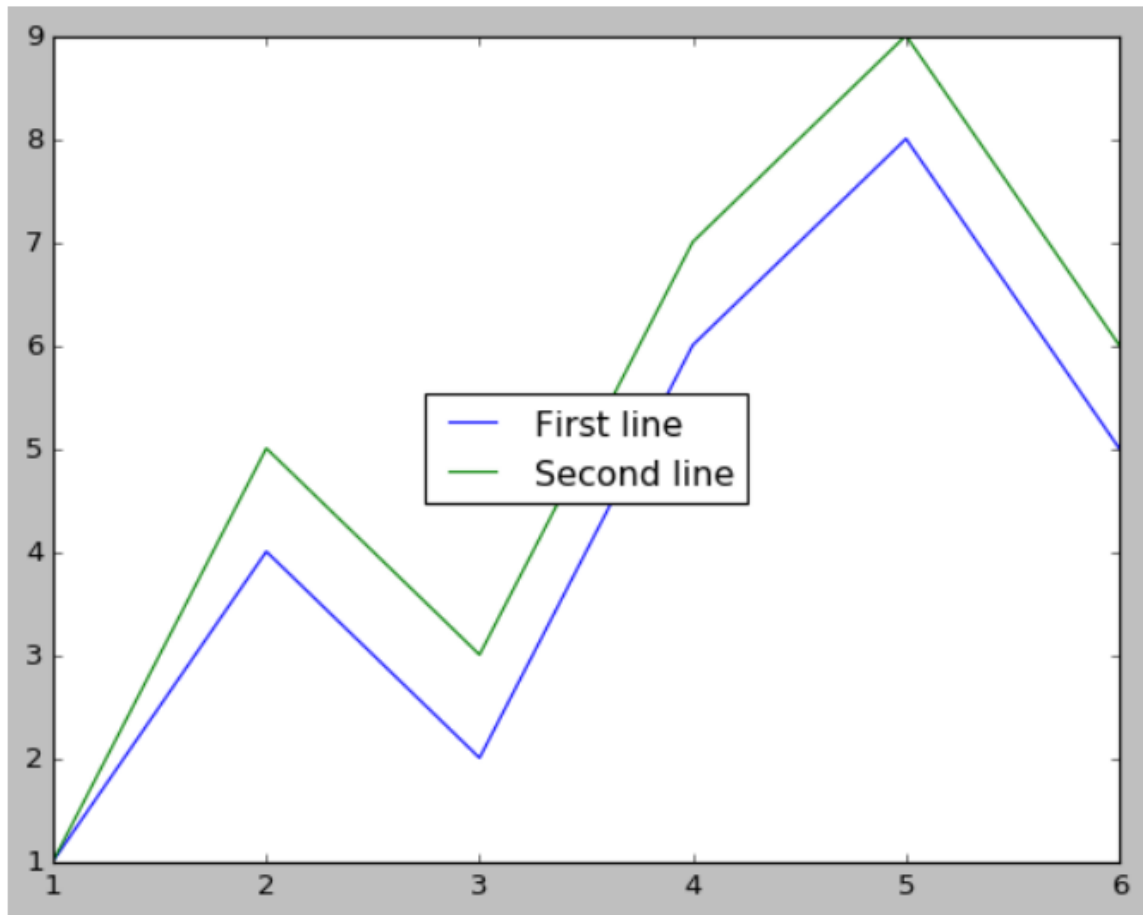


```
import matplotlib.pyplot as plt
plt.style.use('classic')
# Example data
x = [1, 2, 3, 4, 5, 6]
y1 = [1, 4, 2, 6, 8, 5]
y2 = [1, 5, 3, 7, 9, 6]

# Plotting the data
plt.plot(x, y1)
plt.plot(x, y2)

# Adding a legend for existing plot elements
plt.legend(['First line', 'Second line'], loc='center')

# Show the plot
plt.show()
print('Successfully Placed a legend on the Axes...')
```



Successfully Placed a legend on the Axes...

Subplot and Subplot Title

With the `subplot()` function you can draw multiple plots in one figure:

```
import matplotlib.pyplot as plt
import numpy as np

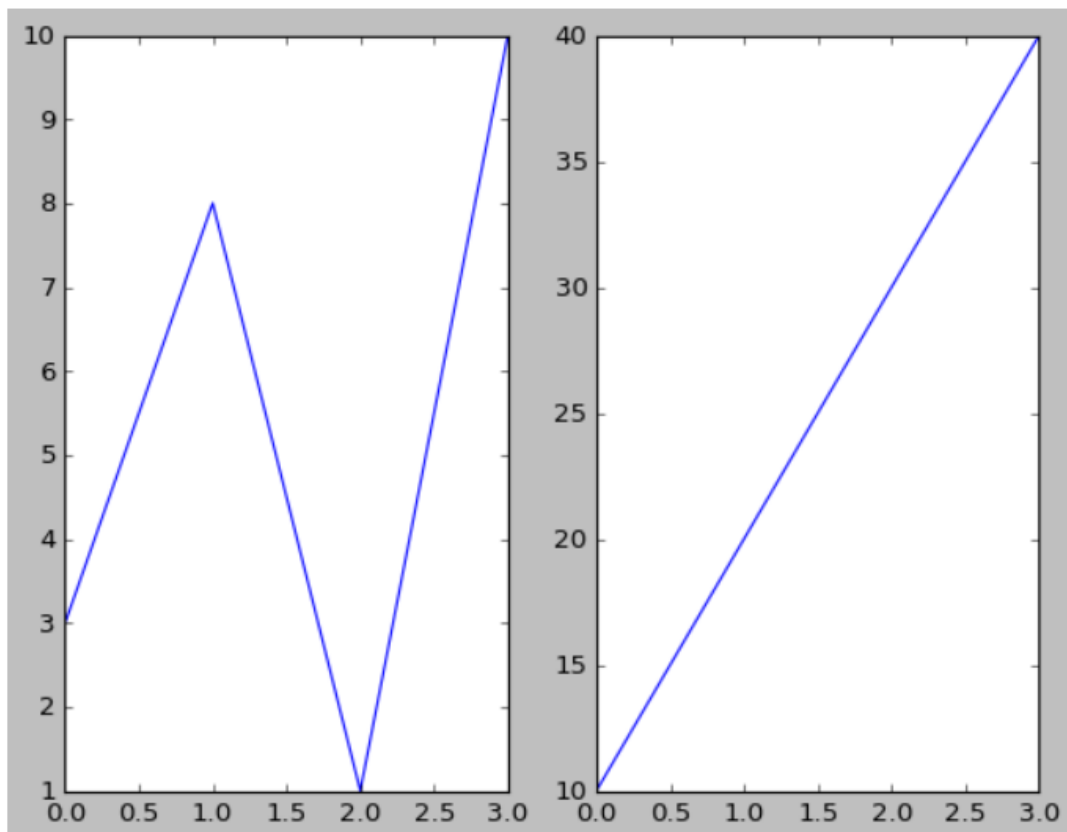
#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)

plt.show()
```



We can add title to each plot and supertitle to the image with title and supitle function:

```
import matplotlib.pyplot as plt
import numpy as np

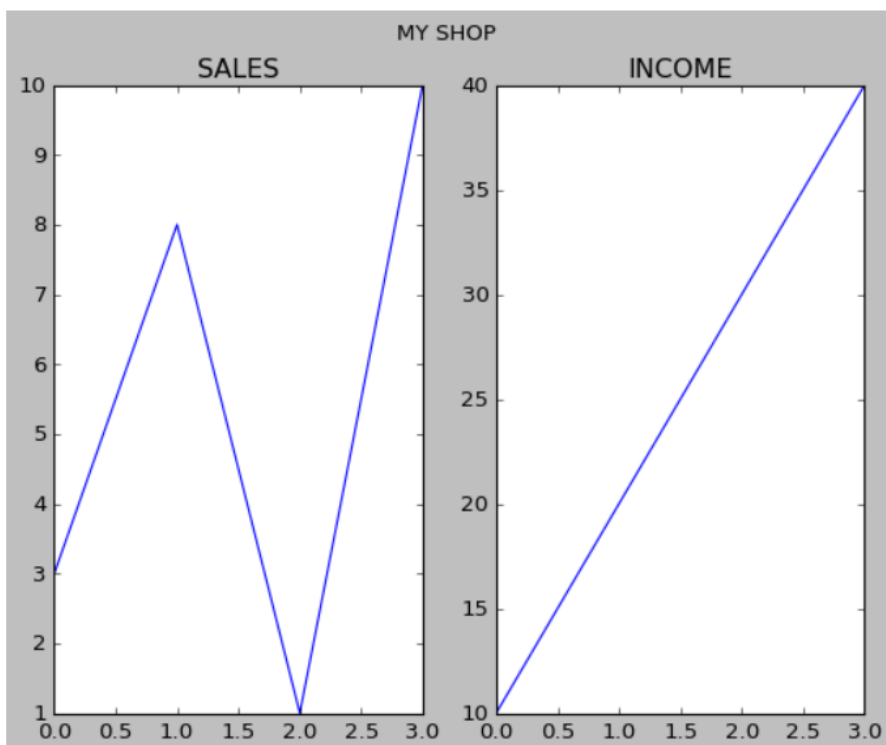
#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)
plt.title("SALES")

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)
plt.title("INCOME")

plt.suptitle("MY SHOP")
plt.show()
```



Subplot Title

The below is the syntax and parameters for setting up the subplot title.

```
ax.set_title('Title')
```

Where,

ax: It represents the axes object of the subplot for which the title is being set.

set_title(): The method used to set the title.

'Title': It is the string representing the text of the title.

```
import matplotlib.pyplot as plt
import numpy as np

# Generating sample data
x = np.linspace(0, 10, 50)
y = np.sin(x)

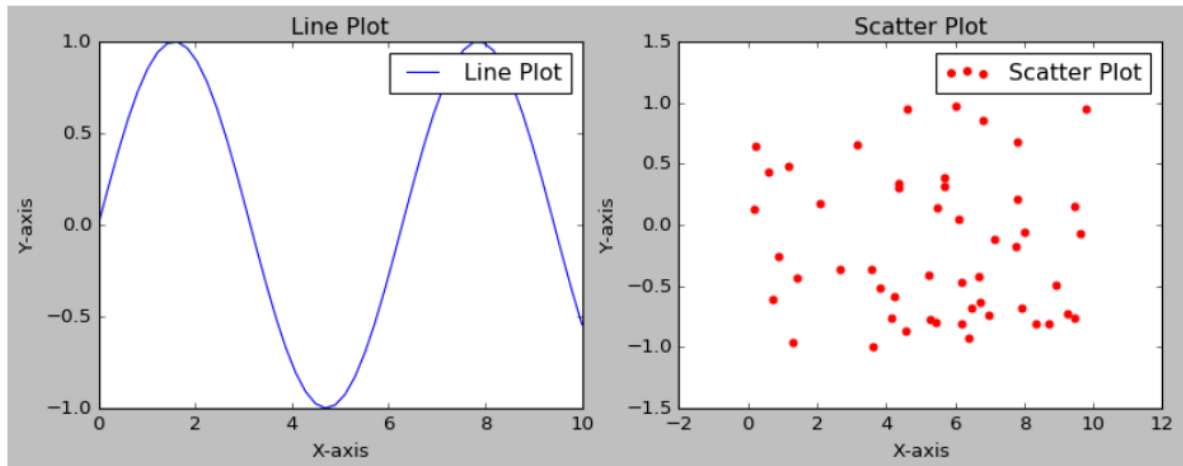
# Generating random data for scatter plot
np.random.seed(0)
x_scatter = np.random.rand(50) * 10
y_scatter = np.random.rand(50) * 2 - 1 # Random values between -1 and 1

# Creating subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4)) # 1 row, 2 columns

# Line plot on the first subplot
ax1.plot(x, y, color='blue', label='Line Plot')
ax1.set_title('Line Plot')
ax1.set_xlabel('X-axis')
ax1.set_ylabel('Y-axis')
ax1.legend()

# Scatter plot on the second subplot
ax2.scatter(x_scatter, y_scatter, color='red', label='Scatter Plot')
ax2.set_title('Scatter Plot')
ax2.set_xlabel('X-axis')
ax2.set_ylabel('Y-axis')
ax2.legend()

# Displaying the subplots
plt.show()
```



Understanding the subplot() Function

The syntax of the function is:

```
matplotlib.pyplot.subplots(nrows=1, ncols=1, *, sharex=False, sharey=False, squeeze=True,
width_ratios=None, height_ratios=None, subplot_kw=None, gridspec_kw=None, **fig_kw)
```

where,

- **nrows, ncols**: Number of rows and columns of the subplot grid.
- **sharex, sharey**: Controls sharing of properties among x or y axes.
- **squeeze**: Controls the squeezing of extra dimensions from the returned array of axes.
- **subplot_kw**: Dictionary with keywords passed to the add_subplot call used to create each subplot.
- **gridspec_kw**: Dictionary with keywords passed to the GridSpec constructor used to create the grid the subplots are placed on.
- Additional keyword arguments are passed to the pyplot.figure() call.


```

import matplotlib.pyplot as plt
import numpy as np

# First create sample data
x = np.linspace(0, 2*np.pi, 400)
y = np.cos(x**2)

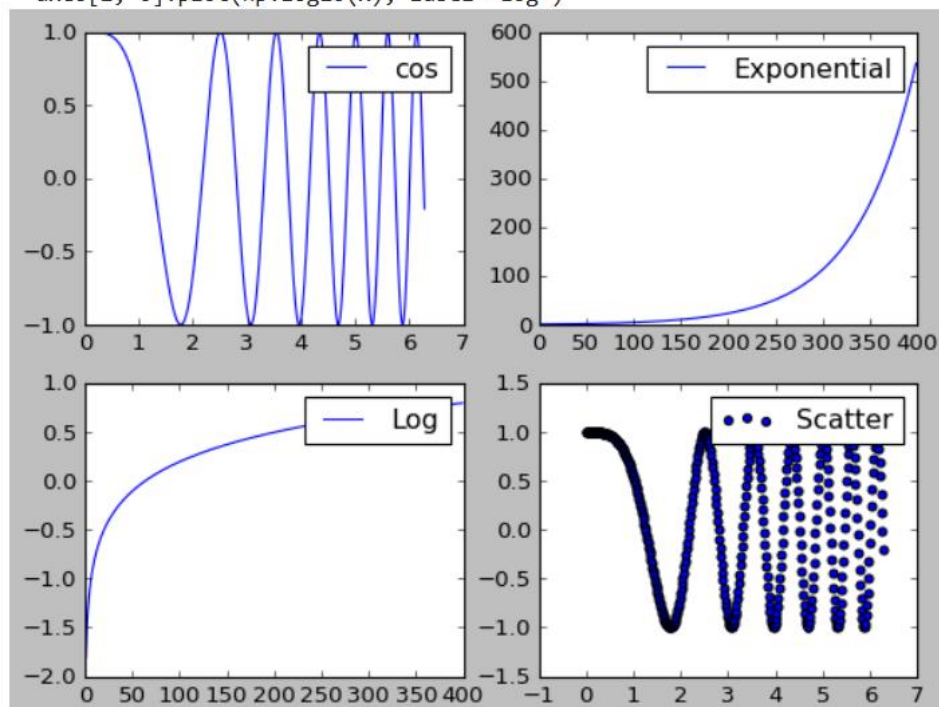
# Create a figure
fig = plt.figure()

# Create a subplot
axes = fig.subplots(2, 2)

axes[0, 0].plot(x, y, label='cos')
axes[0, 0].legend()
axes[0, 1].plot(np.exp(x), label='Exponential')
axes[0, 1].legend()
axes[1, 0].plot(np.log10(x), label='Log')
axes[1, 0].legend()
axes[1, 1].scatter(x, y, label='Scatter')
axes[1, 1].legend()
plt.show()

```

<ipython-input-24-d1c95e2208ec>:18: RuntimeWarning: divide by zero encountered in log10
 axes[1, 0].plot(np.log10(x), label='Log')

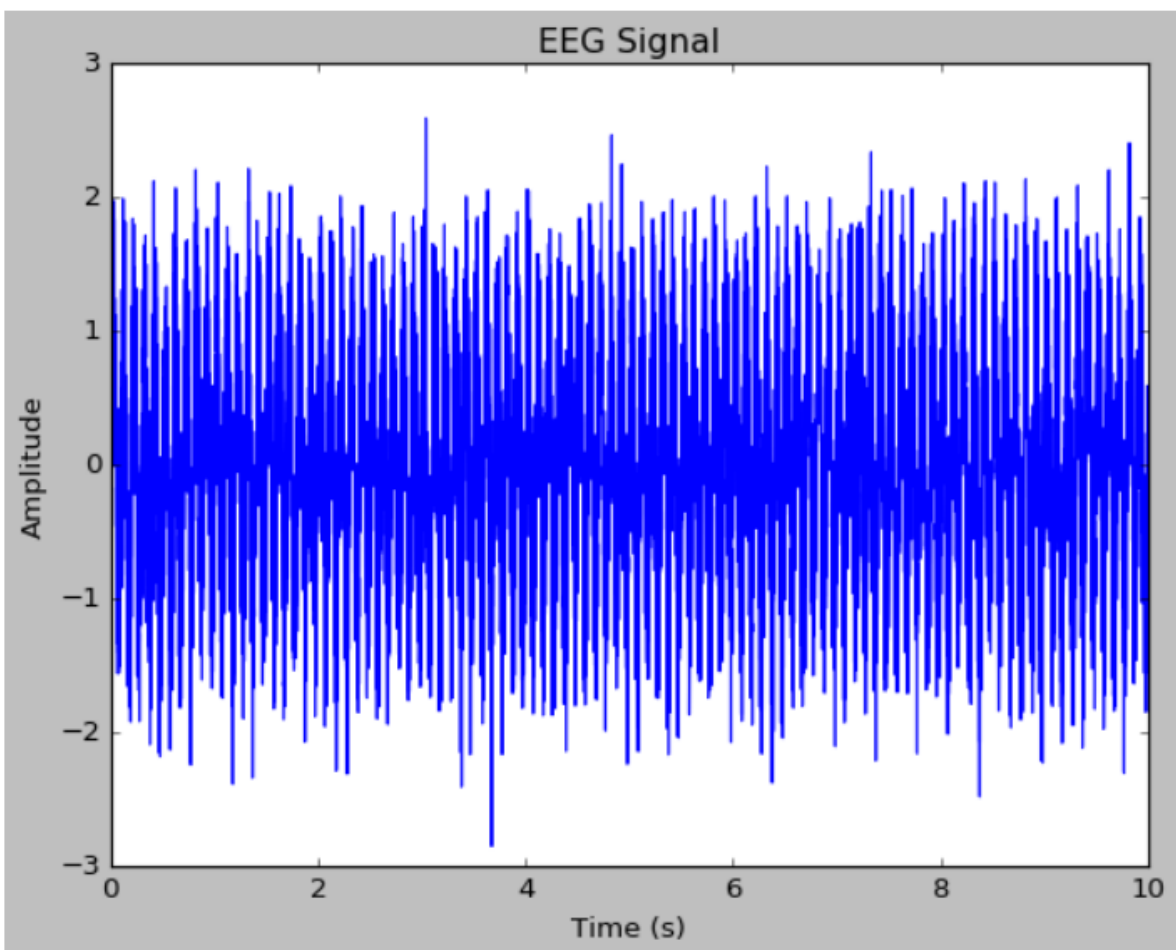


EEG Signal

```
import matplotlib.pyplot as plt
import numpy as np

# Generating sample EEG data
# Sampling frequency (Hz)
fs = 1000
# Time vector (10 seconds)
t = np.arange(0, 10, 1/fs)
# Sample EEG signal
eeg_signal = np.sin(2 * np.pi * 10 * t) + 0.5 * np.random.randn(len(t))

# Plotting EEG signal
plt.plot(t, eeg_signal)
plt.title('EEG Signal')
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.show()
```



Line Plot

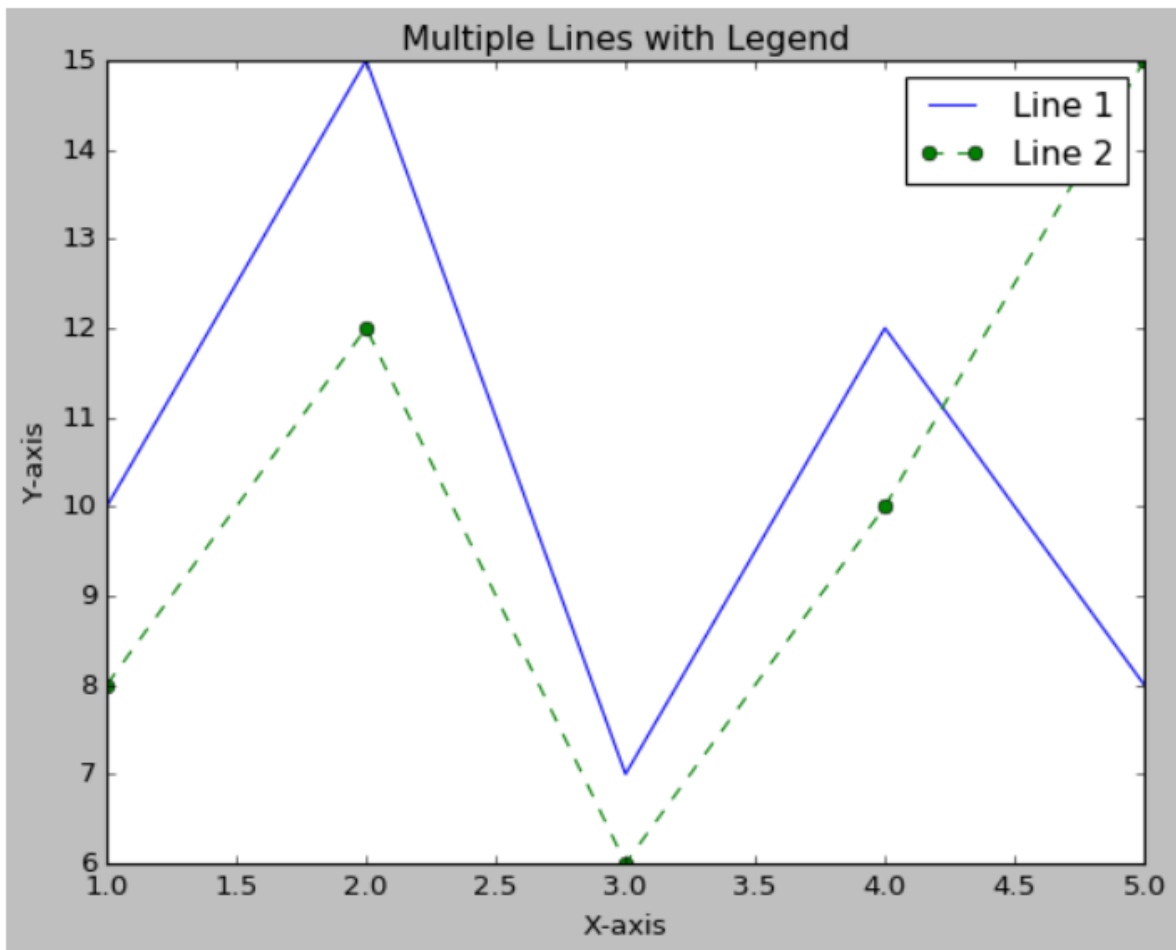
The Syntax for line plot in matplotlib is:

```
matplotlib.pyplot.plot(*args, scalex=True, scaley=True,  
data=None, **kwargs)
```

Where,

- ***args** represents the positional arguments.
- **scalex and scaley** are Boolean values that control whether the x-axis and y-axis should be automatically adjusted to fit the data.
- **data** allows you to pass a DataFrame or similar structure for plotting.
- ****kwargs** represents the additional keyword arguments that allow you to customize the appearance of the plot, such as line style, color, markers, etc.

```
import matplotlib.pyplot as plt  
  
x = [1, 2, 3, 4, 5]  
y1 = [10, 15, 7, 12, 8]  
y2 = [8, 12, 6, 10, 15]  
  
plt.plot(x, y1, label='Line 1')  
plt.plot(x, y2, label='Line 2', linestyle='--', marker='o')  
plt.xlabel('X-axis')  
plt.ylabel('Y-axis')  
plt.title('Multiple Lines with Legend')  
plt.legend()  
plt.show()
```



Area Plots in Matplotlib

In Matplotlib, we can create an area plot using the **fill_between()** function or the **stackplot()** function. These functions allow us to customize colors, transparency, and labels to enhance the visual representation of the data.

The **fill_between()** function shades the area between two specified curves. It is commonly used to highlight specific areas of interest in a plot. The syntax for this function is:

```
fill_between(x, y1, y2=0, where=None, interpolate=False, step=None, **kwargs)
```

Where,

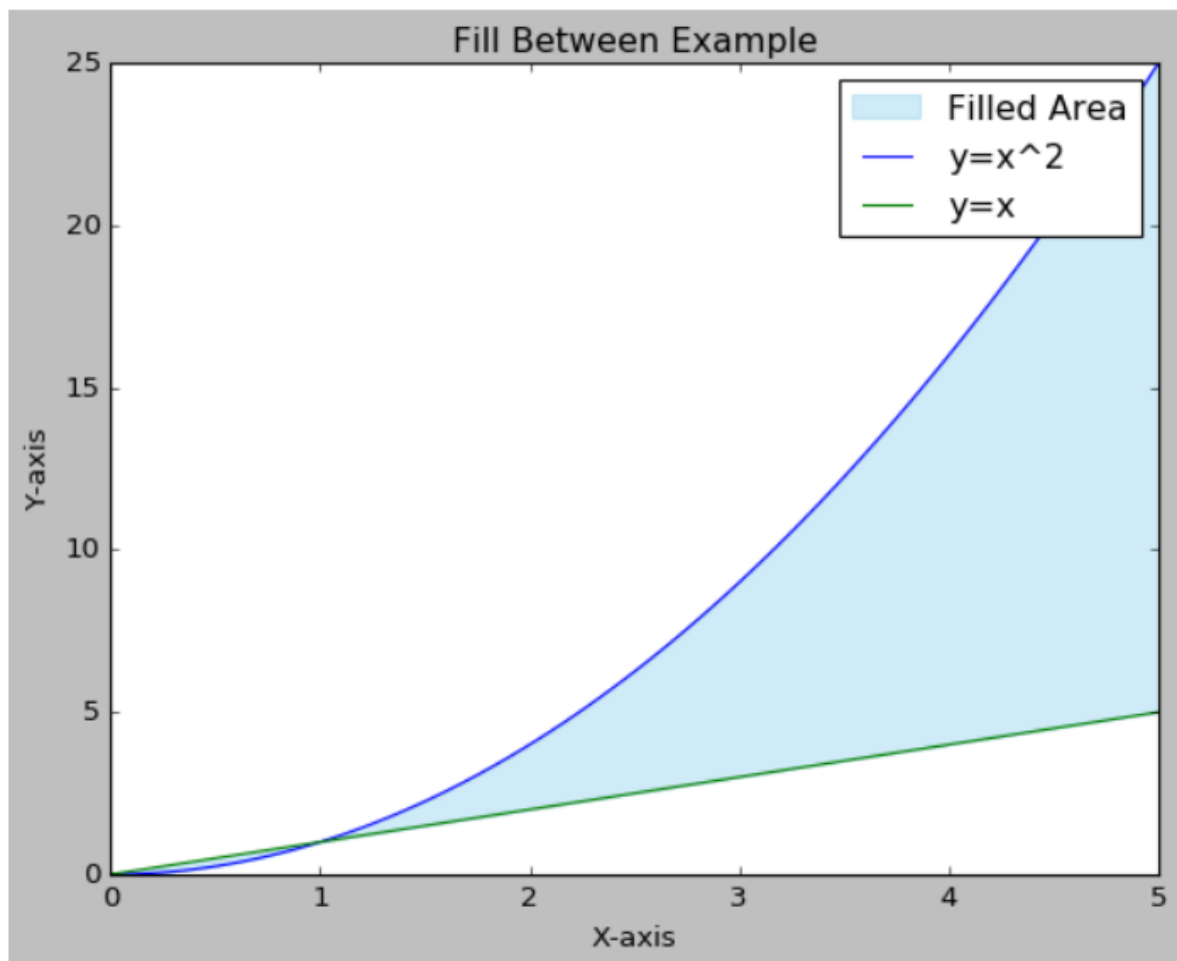
- **x** is the x-coordinates of the data points.
- **y1** is the y-coordinates of the first curve.
- **y2** is the y-coordinates of the second curve or the baseline. If not specified, it defaults to 0, creating a fill between the curve and the x-axis.
- **where** is an optional condition to limit the filling. If provided, only the regions where the condition is True will be filled.
- If **interpolate** is True, the filled area will be interpolated. If False (default), steps are used to create stairs-like filling.
- If **step** is not None, it defines the step type to use for interpolation (plotting). Possible values are 'pre', 'post', or 'mid'.
- ****kwargs** is the additional keyword arguments that control the appearance of the filled region, such as color, alpha, label, etc.

```
import matplotlib.pyplot as plt
import numpy as np
# Generate sample data
x = np.linspace(0, 5, 100)
y1 = x**2
y2 = x

# Fill the region between y1 and y2 with a sky-blue color and set transparency
plt.fill_between(x, y1, y2, color='skyblue', alpha=0.4, label='Filled Area')

# Plot the curves for y=x^2 and y=x
plt.plot(x, y1, label='y=x^2')
plt.plot(x, y2, label='y=x')

# Add labels to the axes
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
# Add a title to the plot
plt.title('Fill Between Example')
# Add a legend to identify the filled area and the curves
plt.legend()
# Show the plot
plt.show()
```



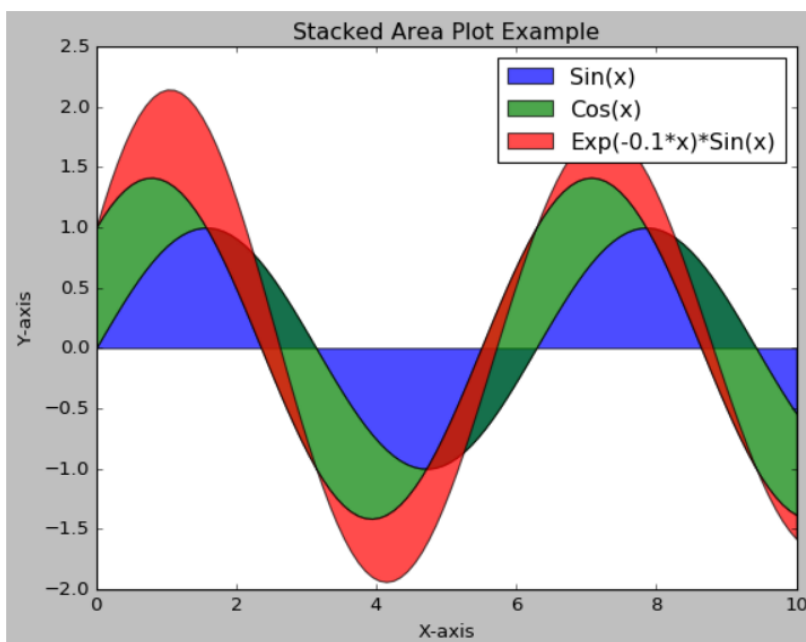
The **stackplot()** function is used to create a stacked area plots. It allows you to represent the cumulative contribution of multiple datasets or categories to a whole over a continuous range. Each category is represented by a filled area, and the combined areas give an overview of the overall trend. The syntax of the function is:

```
stackplot(x, *args, labels=(), colors=None, baseline='zero', alpha=1, **kwargs)
```

Where,

- **x** is the x-coordinates of the data points.
- ***args** is the variable-length argument list of y-coordinates for each dataset or category to be stacked.
- **labels** is a tuple or list of labels for each dataset, used for creating a legend.
- **colors** is a tuple or list of colors for each dataset. If not specified, Matplotlib will automatically choose colors.
- **baseline** specifies whether stacking is relative to 'zero' (default) or 'sym' (symmetric around zero).
- **alpha** specifies the transparency of the filled areas.
- ****kwargs** is the additional keyword arguments that control the appearance of the filled region, such as color, alpha, label, etc.

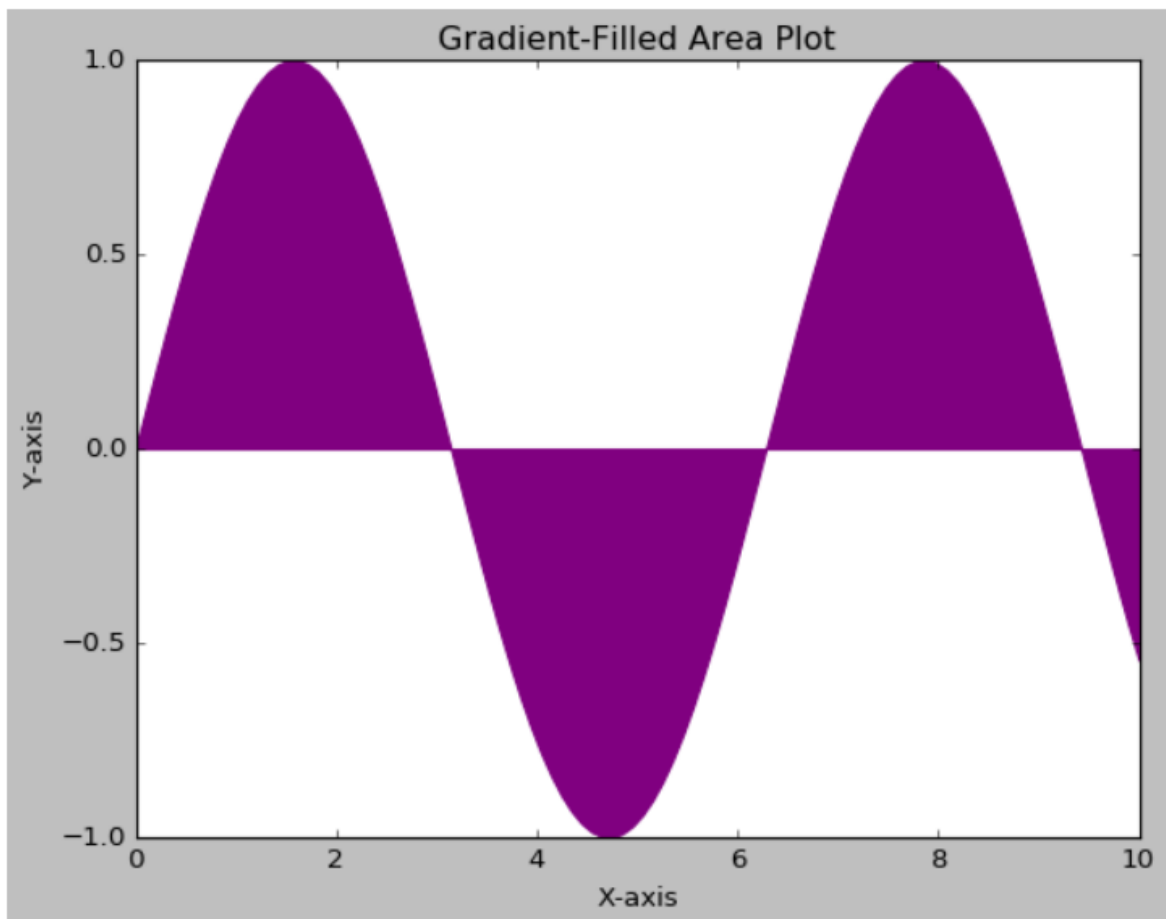
```
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.exp(-0.1 * x) * np.sin(x)
# Create a stacked area plot
plt.stackplot(x, y1, y2, y3, labels=['Sin(x)', 'Cos(x)', 'Exp(-0.1*x)*Sin(x)'], alpha=0.7)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Stacked Area Plot Example')
plt.legend()
plt.show()
```



Gradient-Filled Area Plot

A gradient-filled area plot is like coloring the space under a curve with a smooth transition of colors. It helps emphasize the shape of the curve and how it changes.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 10, 100)
y = np.sin(x)
plt.fill_between(x, y, color='purple', interpolate=True)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Gradient-Filled Area Plot')
plt.show()
```



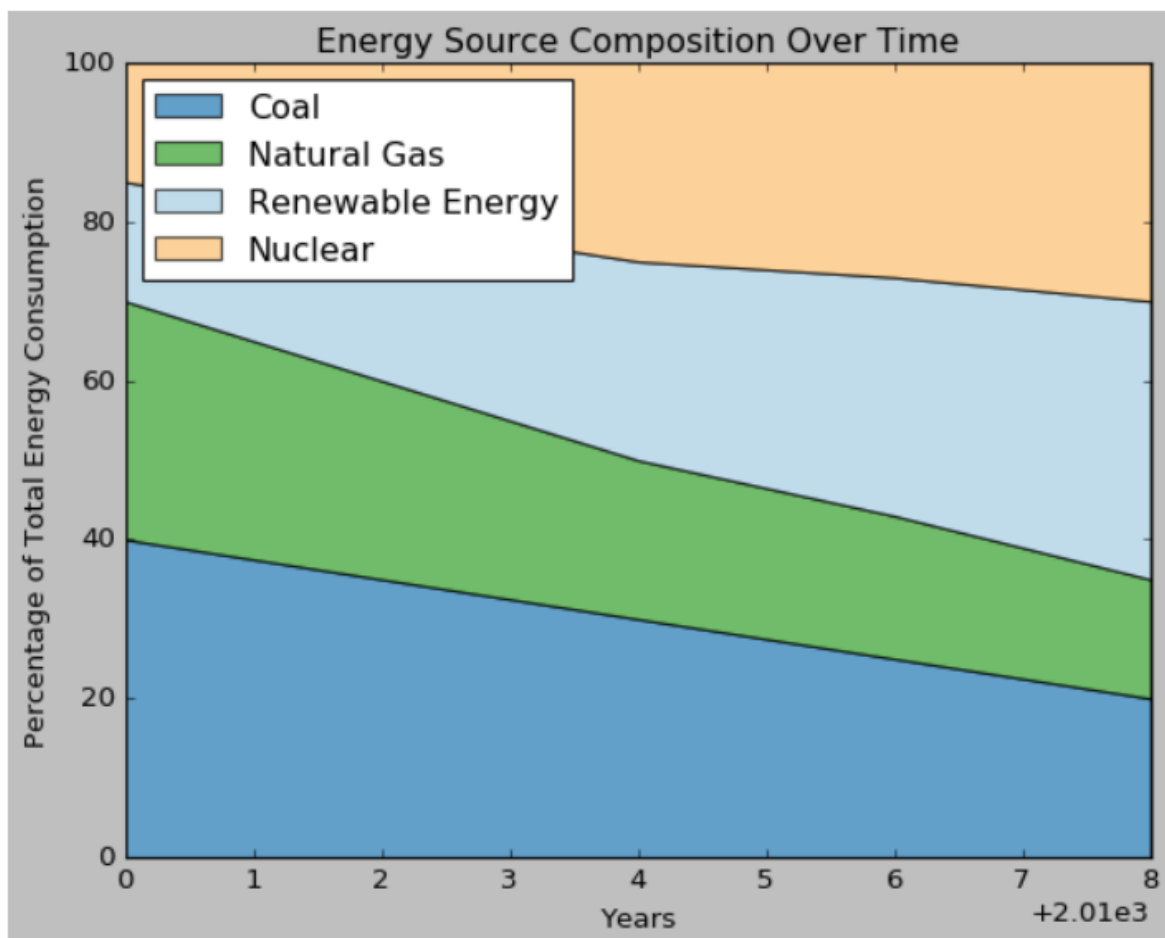
Percentage Stacked Area Plot

```
import matplotlib.pyplot as plt

years = [2010, 2012, 2014, 2016, 2018]
# Percentage contribution of coal to total energy consumption
coal = [40, 35, 30, 25, 20]
# Percentage contribution of natural gas
natural_gas = [30, 25, 20, 18, 15]
# Percentage contribution of renewable energy
renewable_energy = [15, 20, 25, 30, 35]
# Percentage contribution of nuclear energy
nuclear = [15, 20, 25, 27, 30]

plt.stackplot(years, coal, natural_gas, renewable_energy, nuclear,
              labels=['Coal', 'Natural Gas', 'Renewable Energy', 'Nuclear'],
              colors=['#1f78b4', '#33a02c', '#a6cee3', '#fdbf6f'],
              alpha=0.7, baseline='zero')

plt.xlabel('Years')
plt.ylabel('Percentage of Total Energy Consumption')
plt.title('Energy Source Composition Over Time')
plt.legend(loc='upper left')
plt.show()
```



Area Plot with Annotations

An area plot with annotations allows you to add textual or graphical elements to the plot, highlighting specific points or regions on it. It provides additional information about significant data points or regions. These annotations can include labels, arrows, or other markers to draw attention to specific area of interest.

```
import numpy as np
import matplotlib.pyplot as plt

# Simulating monthly sales data
months = np.arange(1, 13)
sales = np.array([10, 15, 12, 18, 25, 30, 28, 35, 32, 28, 20, 15])

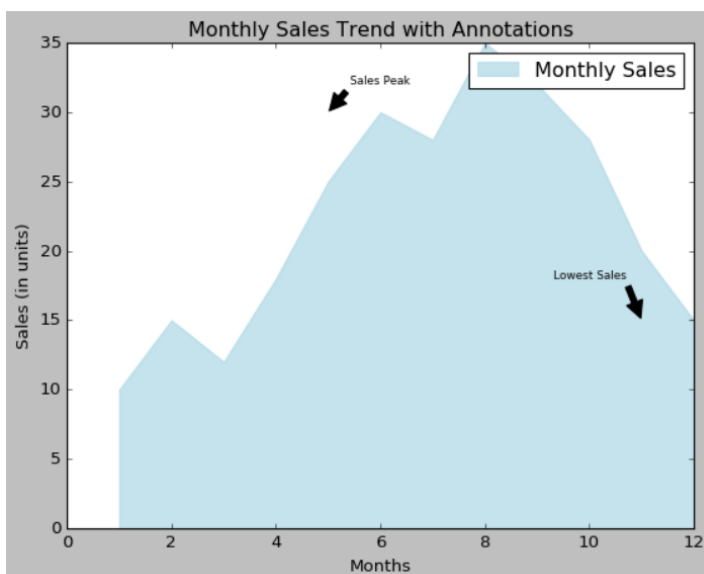
# Create an area plot
plt.fill_between(months, sales, color='lightblue', alpha=0.7, label='Monthly Sales')

# Add annotations for significant points
plt.annotate('Sales Peak', xy=(5, 30), xytext=(6, 32),
            arrowprops=dict(facecolor='black', shrink=0.05),
            fontsize=8, ha='center')

plt.annotate('Lowest Sales', xy=(11, 15), xytext=(10, 18),
            arrowprops=dict(facecolor='black', shrink=0.05),
            fontsize=8, ha='center')

# Add labels and title
plt.xlabel('Months')
plt.ylabel('Sales (in units)')
plt.title('Monthly Sales Trend with Annotations')

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



Bar graphs

The `bar()` function is used to create bar graphs. It takes two main parameters: the positions of the bars on the x-axis and the heights of the bars. The syntax of the `bar()` function is:

```
plt.bar(x, height, width=0.8, align='center', color=None, label=None)
```

Where,

x is the positions of the bars on the x-axis.

height is the heights of the bars.

width (optional) is the width of the bars. Default is 0.8.

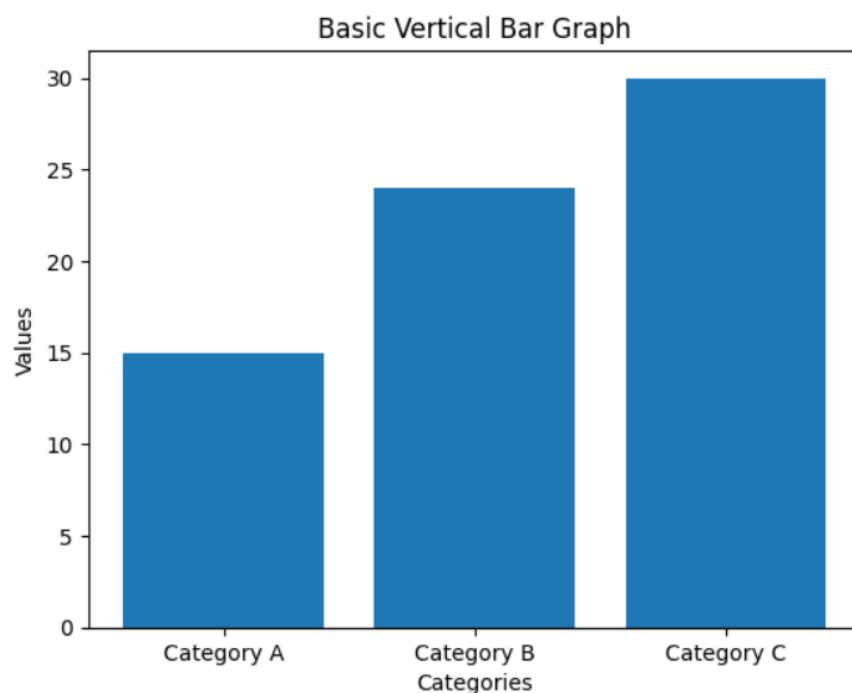
align (optional) is alignment of the bars. Default is 'center'.

color (optional) is the color of the bars. Default is None, which results in the default color.

label (optional) is a label for the legend.

```
import matplotlib.pyplot as plt
categories = ['Category A', 'Category B', 'Category C']
values = [15, 24, 30]

plt.bar(categories, values)
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Basic Vertical Bar Graph')
plt.show()
```

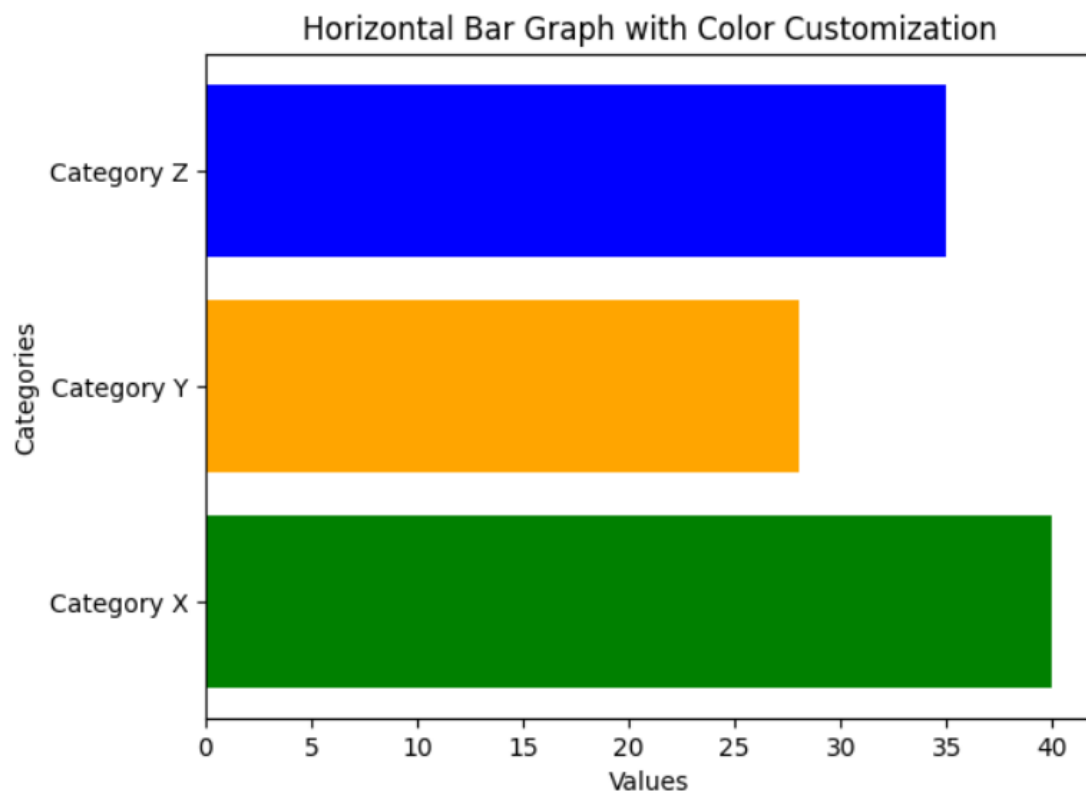


For **Horizontal Plot**,

```
import matplotlib.pyplot as plt

categories = ['Category X', 'Category Y', 'Category Z']
values = [40, 28, 35]

plt.barh(categories, values, color=['green', 'orange', 'blue'])
plt.xlabel('Values')
plt.ylabel('Categories')
plt.title('Horizontal Bar Graph with Color Customization')
plt.show()
```



For Grouped Bar,

```
import matplotlib.pyplot as plt
import numpy as np

# Defining categories and their corresponding values for two groups
categories = ['Category A', 'Category B', 'Category C']
values1 = [15, 24, 30]
values2 = [20, 18, 25]

# Setting the width of the bars
bar_width = 0.35

# Calculating bar positions for both groups
bar_positions1 = np.arange(len(categories))
bar_positions2 = bar_positions1 + bar_width

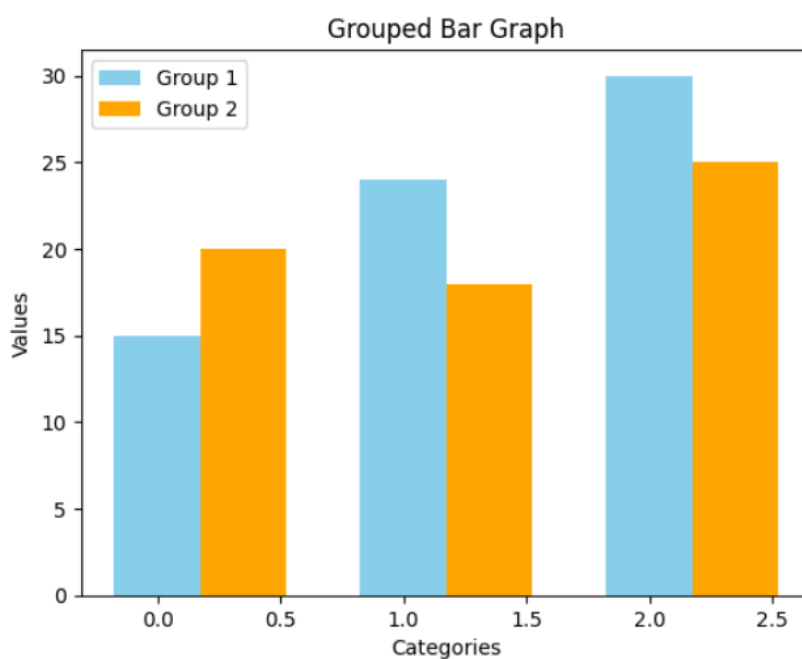
# Creating the first set of bars (Group 1)
plt.bar(bar_positions1, values1, width=bar_width, label='Group 1', color='skyblue')
# Create the second set of bars (Group 2) next to the first set
plt.bar(bar_positions2, values2, width=bar_width, label='Group 2', color='orange')

# Adding labels to the axes
plt.xlabel('Categories')
plt.ylabel('Values')

# Adding a title to the graph
plt.title('Grouped Bar Graph')

# Displaying a legend to identify the groups
plt.legend()

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



For **Stacked Bar graph**,

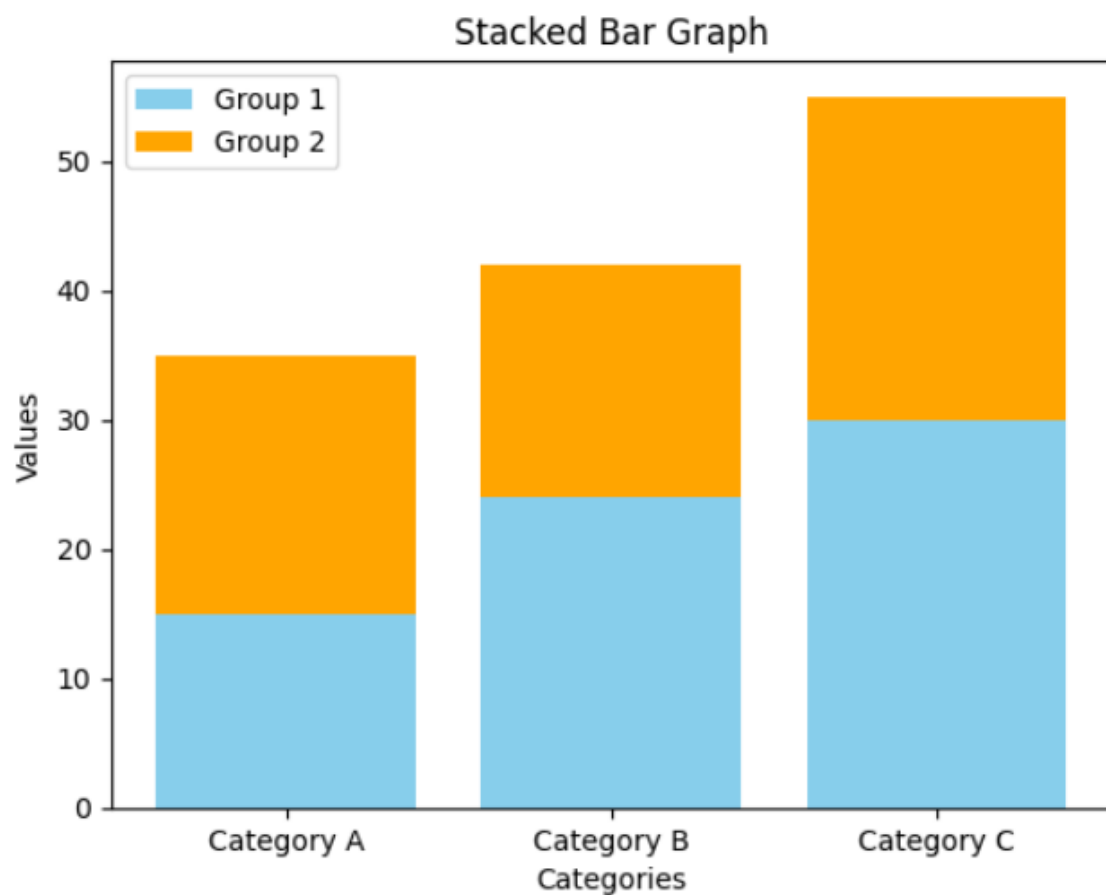
```
import matplotlib.pyplot as plt
# Defining categories and values for two groups
categories = ['Category A', 'Category B', 'Category C']
values1 = [15, 24, 30]
values2 = [20, 18, 25]

# Creating the first set of bars (Group 1) without any offset
plt.bar(categories, values1, label='Group 1', color='skyblue')

# Creating the second set of bars (Group 2) plotted with 'bottom' set to the values of Group 1
# This makes Group 2 bars stacked on top of Group 1 bars
plt.bar(categories, values2, bottom=values1, label='Group 2', color='orange')

# Adding labels to the axes
plt.xlabel('Categories')
plt.ylabel('Values')

plt.title('Stacked Bar Graph')
plt.legend()
plt.show()
```



Histogram

The `hist()` function in Matplotlib takes a dataset as input and divides it into intervals (bins). It then displays the frequency (count) of data points falling within each bin as a bar graph. The syntax for the `hist()` function is:

```
plt.hist(x, bins=None, range=None, density=False, cumulative=False, color=None,
         edgecolor=None, ...)
```

Where,

x is the input data for which the histogram is determined.

bins (optional) is the number of bins or the bin edges.

range (optional) is the lower and upper range of the bins. Default is the minimum and maximum of x

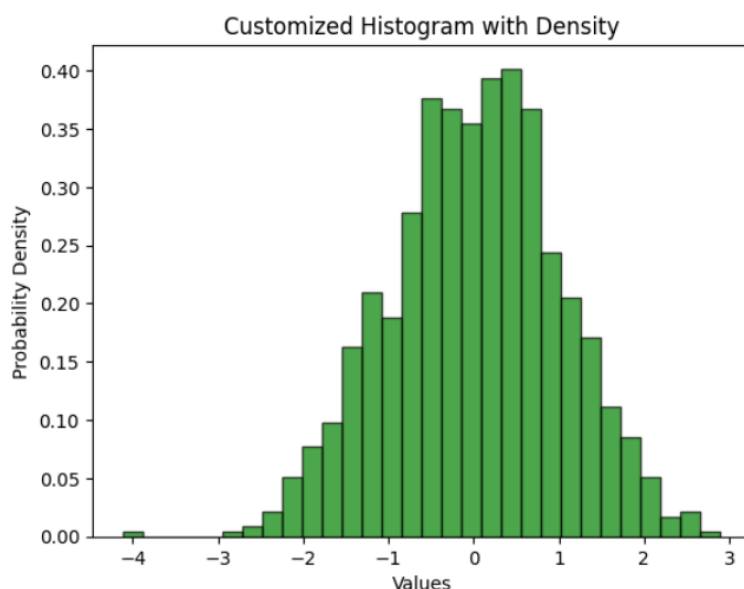
If **density (optional)** is True, the histogram represents a probability density function. Default is False.

If **cumulative (optional)** is True, a cumulative histogram is computed. Default is False.

```
import matplotlib.pyplot as plt
import numpy as np

# Generate random data
data = np.random.randn(1000)

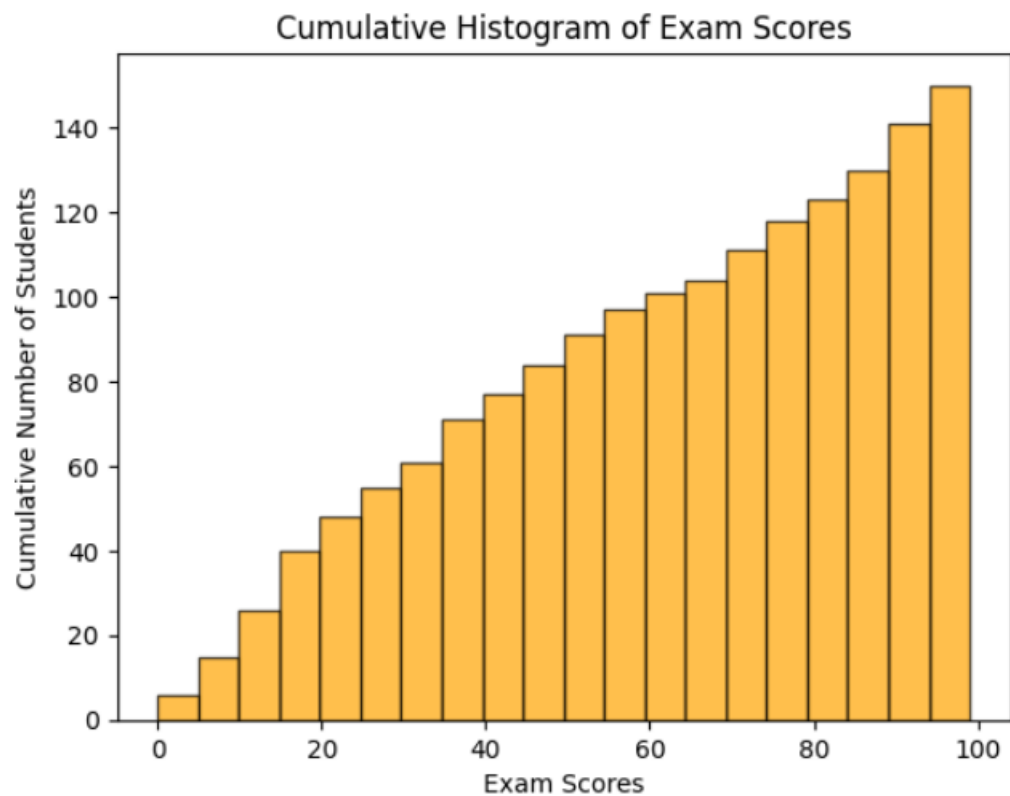
# Create a histogram with density and custom color
plt.hist(data, bins=30, density=True, color='green', edgecolor='black', alpha=0.7)
plt.xlabel('Values')
plt.ylabel('Probability Density')
plt.title('Customized Histogram with Density')
plt.show()
```



```
import matplotlib.pyplot as plt
import numpy as np

# Generate random exam scores (out of 100)
exam_scores = np.random.randint(0, 100, 150)

# Create a cumulative histogram
plt.hist(exam_scores, bins=20, cumulative=True, color='orange', edgecolor='black', alpha=0.7)
plt.xlabel('Exam Scores')
plt.ylabel('Cumulative Number of Students')
plt.title('Cumulative Histogram of Exam Scores')
plt.show()
```



Pie Chart

The **pie()** function in Matplotlib takes an array or a list of values, where each value represents the size of a slice in the pie. The function provides visual proportions and distributions of categorical data in a circular format. The syntax for pie() function is:

```
plt.pie(x, explode=None, labels=None, colors=None, autopct=None, shadow=False, startangle=0)
```

Where,

x is an array or list of values representing the sizes of slices.

explode (optional) is an array or list specifying the fraction of the radius with which to offset each slice.

labels (optional) is the list of strings providing labels for each slice.

colors (optional) is list of colors for each slice.

autopct (optional) format string or function for adding percentage labels on the pie.

If shadow (optional) is True, it adds a shadow to the pie.

startangle (optional) is the angle by which the start of the pie is rotated counter clockwise from the x-axis.

```
import matplotlib.pyplot as plt

# Data for proportions
sizes = [20, 35, 25, 20]
explode = (0, 0.1, 0.1, 0)

# Creating an exploded pie chart
plt.pie(sizes, explode=explode, labels=['Category A', 'Category B', 'Category C', 'Category D'])
plt.title('Exploded Pie Chart')
plt.show()
```

