**Data Visualization using Matplotlib in Python**

**Matplotlib** is a widely-used Python library used for creating static, animated and interactive data visualizations. It is built on the top of **NumPy**and it can easily handles large datasets for creating various types of plots such as line charts, bar charts, scatter plots, etc. These visualizations help us to understand data better by presenting it clearly through graphs and charts. In this article, we will see how to create different types of plots and customize them in matplotlib.

**Visualizing Data with Pyplot using Matplotlib**

Matplotlib provides a module called [pyplot](https://www.geeksforgeeks.org/pyplot-in-matplotlib/" \t "_blank) which offers a **MATLAB-like interface** for creating plots and charts. It simplifies the process of generating various types of visualizations by providing a collection of functions that handle common plotting tasks. Let’s explore some examples with simple code to understand how to use it effectively.

**1. Line Chart**

[**Line chart**](https://www.geeksforgeeks.org/line-chart-in-matplotlib-python/)is one of the basic plots and can be created using the [**plot()**](https://www.geeksforgeeks.org/matplotlib-pyplot-plot-function-in-python/) function. It is used to represent a relationship between two data X and Y on a different axis.

**Syntax:**

*matplotlib.pyplot.plot(x, y, color=None, linestyle='-', marker=None, linewidth=None, markersize=None)*

**Example:**

**import** **matplotlib.pyplot** **as** **plt**

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

plt.plot(x, y)

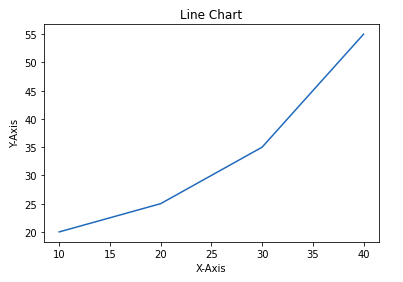
plt.title("Line Chart")

plt.ylabel('Y-Axis')

plt.xlabel('X-Axis')

plt.show()

**Output:**



**2. Bar Chart**

A [bar chart](https://www.geeksforgeeks.org/bar-plot-in-matplotlib/) is a graph that represents the category of data with rectangular bars with lengths and heights which is proportional to the values which they represent. The bar plot can be plotted horizontally or vertically. It describes the comparisons between different categories and can be created using the**bar()** method.

In the below example we will using Pandas library for its implementation on tips dataset. It is the record of the tip given by the customers in a restaurant for two and a half months in the early 1990s and it contains 6 columns. You can download the dataset from [here](https://media.geeksforgeeks.org/wp-content/uploads/20250116111902283403/tip.csv).

**Syntax:**

*matplotlib.pyplot.bar(x, height, width=0.8, bottom=None, color=None, edgecolor=None, linewidth=None)*

**Example:**

**import** **matplotlib.pyplot** **as** **plt**

**import** **pandas** **as** **pd**

data = pd.read\_csv('/content/tip.csv')

x = data['day']

y = data['total\_bill']

plt.bar(x, y)

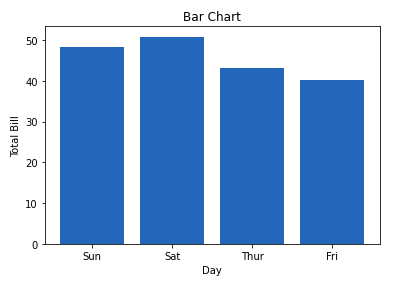
plt.title("Bar chart")

plt.ylabel('Total Bill')

plt.xlabel('Day')

plt.show()

**Output:**



**3. Histogram**

A [**histogram**](https://www.geeksforgeeks.org/plotting-histogram-in-python-using-matplotlib/) is used to represent data provided in a form of some groups. It is a type of bar plot where the X-axis represents the bin ranges while the Y-axis gives information about frequency. The **hist()** function is used to find and create histogram of x.

**Syntax:**

*matplotlib.pyplot.hist(x, bins=None, range=None, density=False, color=None, edgecolor=None, alpha=None)*

**Example:**

**import** **matplotlib.pyplot** **as** **plt**

**import** **pandas** **as** **pd**

data = pd.read\_csv('/content/tip.csv')

x = data['total\_bill']

plt.hist(x)

plt.title("Histogram")

plt.ylabel('Frequency')

plt.xlabel('Total Bill')

plt.show()

**Output:**

Histogram

**4. Scatter Plot**

[**Scatter plots**](https://www.geeksforgeeks.org/matplotlib-pyplot-scatter-in-python/)are used to observe relationships between variables. The **scatter()** method in the matplotlib library is used to draw a scatter plot.

**Syntax:**

*matplotlib.pyplot.scatter(x, y, s=None, c=None, marker=None, linewidths=None, edgecolors=None, alpha=None)*

**Example:**

**import** **matplotlib.pyplot** **as** **plt**

**import** **pandas** **as** **pd**

data = pd.read\_csv('/content/tip.csv')

x = data['day']

y = data['total\_bill']

plt.scatter(x, y)

plt.title("Scatter Plot")

plt.ylabel('Total Bill')

plt.xlabel('Day')

plt.show()

**Output:**

**5. Pie Chart**

[**Pie chart**](https://www.geeksforgeeks.org/plot-a-pie-chart-in-python-using-matplotlib/)is a circular chart used to display only one series of data. The area of slices of the pie represents the percentage of the parts of the data. The slices of pie are called **wedges**. It can be created using the**pie()** method.

**Syntax:**

*matplotlib.pyplot.pie(data, explode=None, labels=None, colors=None, autopct=None, shadow=False)*

**Example:**

**import** **matplotlib.pyplot** **as** **plt**

**import** **pandas** **as** **pd**

data = pd.read\_csv('/content/tip.csv')

cars = ['AUDI', 'BMW', 'FORD',

'TESLA', 'JAGUAR',]

data = [23, 10, 35, 15, 12]

plt.pie(data, labels=cars)

plt.title(" Pie Chart")

plt.show()

**Output:**

**6. Box Plot**

A [Box Plot](https://www.geeksforgeeks.org/box-plot-in-python-using-matplotlib/) is also known as a **Whisker Plot**which is a standardized way of displaying the distribution of data based on a five-number summary: **minimum, first quartile (Q1), median (Q2), third quartile (Q3) and maximum**. It can also show outliers.

**Syntax:**

*matplotlib.pyplot.boxplot(x, notch=False, vert=True, patch\_artist=False, showmeans=False, showcaps=True, showbox=True)*

**Example:**

**import** **matplotlib.pyplot** **as** **plt**

**import** **numpy** **as** **np**

np.random.seed(10)

data = [np.random.normal(0, std, 100) **for** std **in** range(1, 4)]

plt.boxplot(data, vert=**True**, patch\_artist=**True**,

boxprops=dict(facecolor='skyblue'),

medianprops=dict(color='red'))

plt.xlabel('Data Set')

plt.ylabel('Values')

plt.title('Box Plot')

plt.show()

**Output:**

Box Plot

The box shows the [**interquartile range (IQR)**](https://www.geeksforgeeks.org/interquartile-range/)the line inside the box shows the median and the "**whiskers**" extend to the minimum and maximum values within **1.5 \* IQR** from the first and third quartiles. Any points outside this range are considered outliers and are plotted as individual points.

**Objective:**

**Create a box plot using matplotlib with sample data that includes:**

* **Normal distribution**
* **Skewed data**
* **Outliers**
* **Constant values**
* **Small sample size**

**We'll plot all of these in one chart and explain what each shows.**

**✅ Code Example with Explanation:**

**python**

**CopyEdit**

**import matplotlib.pyplot as plt**

**import numpy as np**

**# Set seed for reproducibility**

**np.random.seed(42)**

**# Sample datasets**

**normal\_data = np.random.normal(loc=50, scale=10, size=100) # normal distribution**

**skewed\_data = np.random.exponential(scale=20, size=100) # right-skewed**

**outlier\_data = np.append(np.random.normal(30, 5, 95), [100, 110, 120, 130, 150]) # some large outliers**

**constant\_data = np.full(100, 42) # same value repeated**

**small\_sample = np.random.normal(55, 2, size=5) # very small sample**

**# Combine data for plotting**

**data = [normal\_data, skewed\_data, outlier\_data, constant\_data, small\_sample]**

**# Box plot**

**plt.figure(figsize=(12, 6))**

**plt.boxplot(data, patch\_artist=True, labels=[**

**"Normal", "Skewed", "Outliers", "Constant", "Small Sample"**

**])**

**plt.title("Box Plot Examples for Data Preprocessing")**

**plt.ylabel("Value Range")**

**plt.grid(True)**

**plt.show()**

**📊 Output Explanation**

**1. Normal Data**

* **Symmetric box**
* **Median in center**
* **Whiskers are nearly equal**
* **No/few outliers**

**✅ Indicates clean, well-behaved data.**

**2. Skewed Data**

* **Longer upper whisker**
* **Median pushed downward**
* **Often indicates log-normal or exponential distributions**

**⚠️ Might need log transformation or standardization.**

**3. Outlier Data**

* **Many points shown as dots beyond whiskers**
* **Whiskers are short, box is small**

**⚠️ Consider removing or treating outliers via z-score, IQR filtering.**

**4. Constant Data**

* **Box is flat (height = 0)**
* **No whiskers, no spread**

**⚠️ Feature is non-informative → drop it during preprocessing.**

**5. Small Sample**

* **Box/whiskers may look odd or minimal**
* **Hard to assess distribution**

**⚠️ Small datasets may need bootstrapping or augmentation.**

**✅ Summary**

| **Pattern** | **What It Suggests** | **Preprocessing Tips** |
| --- | --- | --- |
| **Outliers** | **Extreme values affect models** | **Remove or transform them** |
| **Skewness** | **Distribution not symmetric** | **Try log, sqrt, or Box-Cox transformation** |
| **No variance** | **All values same** | **Drop that feature** |
| **Small samples** | **Too few data points** | **Augment or collect more data** |

**7. Heatmap**

A [Heatmap](https://www.geeksforgeeks.org/generate-a-heatmap-in-matplotlib-using-a-scatter-dataset/) represents data in a matrix form where individual values are represented as colors. They are useful for visualizing the magnitude of multiple features in a two-dimensional surface and identifying patterns, correlations and concentrations.

**Syntax:**

*matplotlib.pyplot.imshow(X, cmap=None, interpolation=None, aspect=None)*

**Example:**

import matplotlib.pyplot as plt

import numpy as np

​

np.random.seed(0)

data = np.random.rand(10, 10)

​

plt.imshow(data, cmap='viridis', interpolation='nearest')

plt.colorbar()

​

plt.xlabel('X-axis Label')

plt.ylabel('Y-axis Label')

plt.title('Heatmap')

​

plt.show()

**Output:**

**cmap='viridis' — Color Map**

**✅ What it is:**

cmap stands for **colormap**. It determines how numeric values are mapped to colors in a plot.

**🔹 'viridis':**

* The **default colormap** in Matplotlib (since v2.0)
* Smooth transition from **dark purple → blue → green → yellow**
* **Perceptually uniform** (easy on eyes, colorblind-friendly)

**📊 Example:**

python

CopyEdit

plt.imshow(data, cmap='viridis')

* Low values → dark purple
* High values → bright yellow

**✅ Other Common Colormaps:**

| **Name** | **Description** | **Use Case** |
| --- | --- | --- |
| 'plasma' | Purple → yellow | Good for visual appeal |
| 'coolwarm' | Blue → white → red | Ideal for **divergent** data (e.g., correlation) |
| 'gray' | Black → white | Greyscale, useful for images |
| 'inferno', 'magma' | Visually pleasing dark maps | Heat-style plots |

**🔧 interpolation='nearest'**

**✅ What it does:**

Determines **how pixels are scaled** when the image is enlarged (i.e., interpolated between grid points).

**🔹 'nearest':**

* **No smoothing**; each cell is a sharp square (nearest neighbor)
* Best when you want **clear pixel boundaries** (like in a heatmap or matrix)

**📊 Example:**

python

CopyEdit

plt.imshow(data, cmap='viridis', interpolation='nearest')

* Zooms in on the data without blurring
* Each number/region stays **visually crisp**

**✅ Other interpolation options:**

| **Option** | **Description** |
| --- | --- |
| 'nearest' | No smoothing, blocky look (best for categorical or matrix) |
| 'bilinear' | Soft smoothing between pixels |
| 'bicubic' | Even smoother (slightly slower) |
| 'hanning', 'lanczos', etc. | Advanced smoothing filters |

**🔍 Side-by-side (visually):**

| **Option** | **Visual Effect** | **Use When...** |
| --- | --- | --- |
| interpolation='nearest' | ![Blocky] | You want each pixel to be distinct (e.g., a confusion matrix) |
| interpolation='bilinear' | ![Smooth] | You're visualizing continuous data or images |

**Heatmap**

The color bar on the side provides a scale to interpret the colors, darker colors representing lower values and lighter colors representing higher values. This type of plot is used in fields like data analysis, bioinformatics and finance to visualize data correlations and distributions across a matrix.

**What Is a Heatmap?**

A **heatmap** is a grid-based color representation of numerical values. Each cell’s color reflects its magnitude.

**✅ Common Uses of Heatmaps:**

| **Use Case** | **Description** |
| --- | --- |
| **Correlation Matrix** | Show relationship between variables in a dataset |
| **Confusion Matrix** | Visualize true vs predicted classes in classification problems |
| **Feature Intensity** | Show how different features vary over time, locations, or samples |
| **Missing Value Visualization** | Highlight missing or anomalous values in a dataset |

**✅ Example: Heatmap of a Correlation Matrix (using matplotlib + seaborn)**

import numpy as np

import seaborn as sns

import matplotlib.pyplot as plt

import pandas as pd

# Sample dataset

np.random.seed(42)

data = pd.DataFrame({

'Math': np.random.randint(50, 100, size=20),

'Physics': np.random.randint(40, 90, size=20),

'Chemistry': np.random.randint(45, 95, size=20),

'English': np.random.randint(60, 85, size=20)

})

# Compute correlation matrix

corr\_matrix = data.corr()

# Plot heatmap

plt.figure(figsize=(8, 6))

sns.heatmap(corr\_matrix, annot=True, cmap='coolwarm', linewidths=0.5, fmt=".2f")

plt.title("Correlation Heatmap of Subjects")

plt.show()

**📊 Output Explanation:**

* annot=True: shows the correlation values in the cells.
* cmap='coolwarm': color gradient from blue (negative) to red (positive).
* linewidths=0.5: adds gridlines between cells.
* corr() computes the **Pearson correlation coefficient**.

**🔍**If Math and Physics show 0.85, they increase together → **highly correlated**.

**data.corr() in Pandas:**

data.corr() computes the **pairwise correlation** of **numeric columns** in a DataFrame.

**🔍 What is Correlation?**

**Correlation** measures the **linear relationship** between two variables. It ranges from:

* +1.0 → Perfect positive correlation (both go up together)
* 0.0 → No linear correlation
* -1.0 → Perfect negative correlation (one goes up, the other goes down)

**🧪 Example:**

import pandas as pd

import numpy as np

# Create sample DataFrame

data = pd.DataFrame({

'Math': [80, 90, 85, 70, 60],

'Physics': [82, 88, 84, 65, 63],

'English': [60, 65, 62, 85, 90]

})

# Compute correlation matrix

corr\_matrix = data.corr()

print(corr\_matrix)

**Output:**

Math Physics English

Math 1.000 0.987 -0.946

Physics 0.987 1.000 -0.924

English -0.946 -0.924 1.000

**✅ Interpretation:**

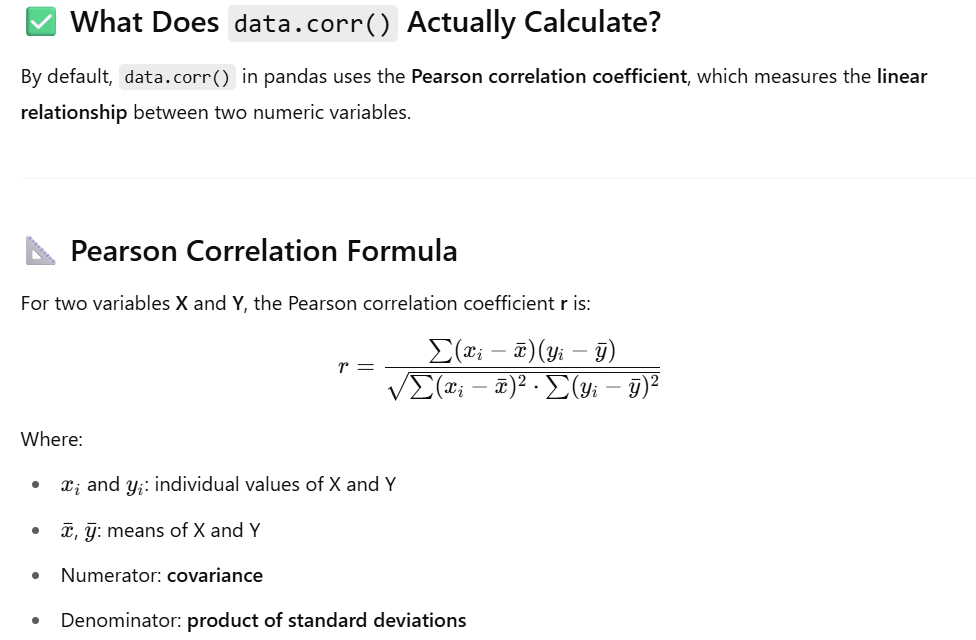
* Math vs Physics: 0.987 → very strong positive correlation
* Math vs English: -0.946 → strong negative correlation
* Diagonal is always 1.0 (correlation with itself)

**📦 Correlation Methods (optional argument in data.corr(method=...)):**

| **Method** | **Meaning** |
| --- | --- |
| 'pearson' (default) | Measures linear relationship |
| 'kendall' | Measures rank-based correlation |
| 'spearman' | Measures monotonic (ordered) relationship |

**✅ Real Use Cases:**

| **Use Case** | **Why Use data.corr()** |
| --- | --- |
| **Feature selection** | Remove highly correlated features (multicollinearity) |
| **EDA (Exploratory Data Analysis)** | Understand variable relationships |
| **Visualizations (heatmap)** | Make it easier to spot patterns |



**🔥 With Heatmap:**

python

CopyEdit

import seaborn as sns

import matplotlib.pyplot as plt

sns.heatmap(data.corr(), annot=True, cmap='coolwarm')

plt.title("Correlation Matrix")

plt.show()

Let me know if you'd like to:

* Calculate correlation only between selected columns
* Drop columns based on high correlation
* Use this in machine learning preprocessing

**✅ Other Heatmap Examples**

**1. Heatmap from a 2D Numpy Array**

arr = np.random.rand(5, 5) # 5x5 matrix

sns.heatmap(arr, annot=True, cmap="YlGnBu")

plt.title("Random Value Heatmap")

plt.show()

**2. Visualizing Missing Data**

import seaborn as sns

import pandas as pd

import numpy as np

df = pd.DataFrame({

"A": [1, 2, np.nan, 4],

"B": [np.nan, 2, 3, 4],

"C": [1, np.nan, 3, 4]

})

sns.heatmap(df.isnull(), cmap='Reds', cbar=False)

plt.title("Missing Value Heatmap")

plt.show()

**✅ Summary: When to Use Heatmaps**

| **Use It For...** | **Why?** |
| --- | --- |
| **Correlation analysis** | Easily spot strong/weak relationships |
| **Confusion matrix** | Model performance summary |
| **Data profiling/preprocessing** | Detect anomalies, patterns, missing data |
| **Feature comparison** | Intensity mapping of values (e.g., gene data) |

Let me know if you want:

* To build a heatmap from real-world data (like Titanic or Iris)
* A 3D heatmap
* Exporting heatmaps to files for reports

**How to Customize Matplotlib Visualizations?**

Matplotlib allows many ways for [customization and styling](https://www.geeksforgeeks.org/python-matplotlib-an-overview/)of our plots. We can change colors, add labels, adjust styles and much more. By applying these customization techniques to basic plots we can make our visualizations clearer and more informative. Lets see various customizing ways:

**1. Customizing Line Chart**

We can customize line charts using these properties:

1. **Color:**Change the color of the line
2. **Linewidth**: Adjust the width of the line
3. **Marker**: Change the style of plotted points
4. **Markersize**: Change the size of the markers
5. **Linestyle**: Define the style of the line like solid, dashed, etc.

**Example:**

**import** **matplotlib.pyplot** **as** **plt**

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

plt.plot(x, y, color='green', linewidth=3, marker='o',

markersize=15, linestyle='--')

plt.title("Customizing Line Chart")

plt.ylabel('Y-Axis')

plt.xlabel('X-Axis')

plt.show()

**Output:**

Customizing Line Chart

**2. Customizing Bar Chart**

Bar charts can be made more informative and visually appealing by customizing:

* **Color:** Fill color of the bars
* **Edgecolor:** Color of the bar edges
* **Linewidth:**Thickness of the edges
* **Width:**Width of each bar

**import** **matplotlib.pyplot** **as** **plt**

**import** **pandas** **as** **pd**

data = pd.read\_csv('/content/tip.csv')

x = data['day']

y = data['total\_bill']

plt.bar(x, y, color='green', edgecolor='blue',

linewidth=2)

plt.title("Customizing Bar Chart")

plt.ylabel('Total Bill')

plt.xlabel('Day')

plt.show()

**Output:**

Customizing Bar Chart

The lines between bars correspond to the values on the Y-axis for each X-axis category.

**3. Customizing Histogram Plot**

To make histogram plots more effective we can apply various customizations:

* **Bins:**Number of groups (bins) to divide data into
* **Color**: Bar fill color
* **Edgecolor**: Bar edge color
* **Linestyle**: Style of the edges like solid, dashed, etc.
* **Alpha**: Transparency level (0 = transparent, 1 = opaque)

**Example:**

**import** **matplotlib.pyplot** **as** **plt**

**import** **pandas** **as** **pd**

data = pd.read\_csv('/content/tip.csv')

x = data['total\_bill']

plt.hist(x, bins=25, color='green', edgecolor='blue',

linestyle='--', alpha=0.5)

plt.title(" Customizing Histogram Plot")

plt.ylabel('Frequency')

plt.xlabel('Total Bill')

plt.show()

**Output:**

Customizing Histogram Plot

**4. Customizing Scatter Plot**

Scatter plots can be enhanced with:

* **S:** Marker size (single value or array)
* **C:** Color of markers or sequence of colors
* **Marker:** Marker style like circle, diamond, etc.
* **Linewidths:** Width of marker borders
* **Edgecolor:** Color of marker borders
* **Alpha:** Blending value, between 0 (transparent) and 1 (opaque)

**import** **matplotlib.pyplot** **as** **plt**

**import** **pandas** **as** **pd**

data = pd.read\_csv('/content/tip.csv')

x = data['day']

y = data['total\_bill']

plt.scatter(x, y, c=data['size'], s=data['total\_bill'],

marker='D', alpha=0.5)

plt.title("Customizing Scatter Plott")

plt.ylabel('Total Bill')

plt.xlabel('Day')

plt.show()

**Output:**

Customizing Scatter Plot

**5. Customizing Pie Chart**

To make our pie charts more effective and visually appealing we consider the following customization:

* **Explode:**Moving the wedges of the plot
* **Autopct:**Label the wedge with their numerical value.
* **Color:** Colors of the slices
* **Sadow:** Used to create a shadow effect

**Example:**

**import** **matplotlib.pyplot** **as** **plt**

**import** **pandas** **as** **pd**

data = pd.read\_csv('/content/tip.csv')

cars = ['AUDI', 'BMW', 'FORD',

'TESLA', 'JAGUAR',]

data = [23, 13, 35, 15, 12]

explode = [0.1, 0.5, 0, 0, 0]

colors = ( "orange", "cyan", "yellow",

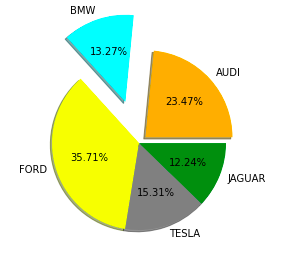
"grey", "green",)

plt.pie(data, labels=cars, explode=explode, autopct='**%1.2f%%**',

colors=colors, shadow=**True**)

plt.show()

**Output:**



**Matplotlib’s Core Components: Figures and Axes**

Before we proceed let’s understand two classes which are important for working with Matplotlib.

**1. Figure class**

The [figure class](https://www.geeksforgeeks.org/matplotlib-figure-figure-in-python/) is like the entire canvas or window where all plots are drawn. Think of it as the overall page or frame that can contain one or more plots. We can create a Figure using the [**figure()**](https://www.geeksforgeeks.org/matplotlib-pyplot-figure-in-python/) function. It controls the size, background color and other properties of the whole drawing area.

**Syntax:**

*matplotlib.figure.Figure(figsize=None, dpi=None, facecolor=None, edgecolor=None, linewidth=0.0, ...)*

**Example:**

**import** **matplotlib.pyplot** **as** **plt**

**from** **matplotlib.figure** **import** Figure

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

fig = plt.figure(figsize =(7, 5), facecolor='g',

edgecolor='b', linewidth=7)

ax = fig.add\_axes([1, 1, 1, 1])

ax.plot(x, y)

plt.title("Linear graph", fontsize=25, color="yellow")

plt.ylabel('Y-Axis')

plt.xlabel('X-Axis')

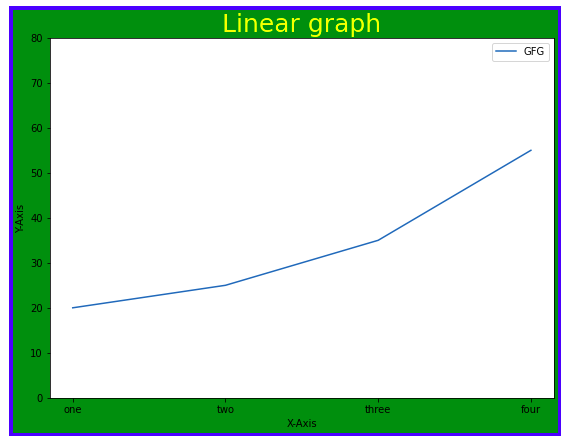
plt.ylim(0, 80)

plt.xticks(x, labels=["one", "two", "three", "four"])

plt.legend(["GFG"])

plt.show()

**Output:**



**2. Axes Class**

[**Axes class**](https://www.geeksforgeeks.org/matplotlib-axes-class/) represents the actual plotting area where data is drawn. It is the most basic and flexible for creating plots or subplots within a figure. A single figure can contain multiple axes but each Axes object belongs to only one figure. We can create an Axes object using the [axes()](https://www.geeksforgeeks.org/matplotlib-pyplot-axes-in-python/) function.

**Syntax:**

*axes([left, bottom, width, height])*

Like **pyplot**, the Axes class provides methods to customize our plot which includes:

* **ax.set\_title()**: Add a title to the plot
* **ax.set\_xlabel(), ax.set\_ylabel()**: Add labels to the X and Y axes
* **ax.set\_xlim(), ax.set\_ylim()**: Set limits for the axes
* **ax.set\_xticklabels(), ax.set\_yticklabels()**: Customize tick labels
* **ax.legend()**: Add a legend to describe plot elements

**Example:**

**import** **matplotlib.pyplot** **as** **plt**

**from** **matplotlib.figure** **import** Figure

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

fig = plt.figure(figsize = (5, 4))

ax = fig.add\_axes([1, 1, 1, 1])

ax1 = ax.plot(x, y)

ax2 = ax.plot(y, x)

ax.set\_title("Linear Graph")

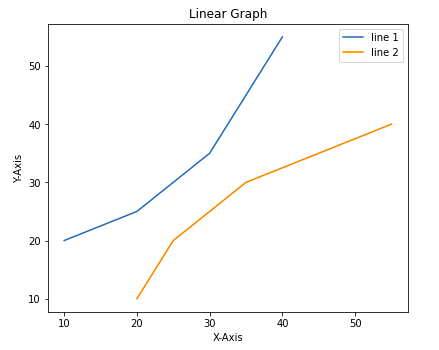
ax.set\_xlabel("X-Axis")

ax.set\_ylabel("Y-Axis")

ax.legend(labels = ('line 1', 'line 2'))

plt.show()

**Output:**



**Advanced Techniques for Visualizing Subplots**

We have learned how to add basic parts to a graph to show more information. One method can be by calling the plot function again and again with a different set of values as shown in the above example. Now let’s see how to draw multiple graphs in one figure using some Matplotlib functions and how to create subplots.

**Method 1: Using add\_axes()**

The [add\_axes() method](https://www.geeksforgeeks.org/matplotlib-figure-figure-add_axes-in-python/" \t "_blank) allows us to manually add axes to a figure in Matplotlib. It takes a list of four values **[left, bottom, width, height]** to specify the position and size of the axes.

**Example:**

**import** **matplotlib.pyplot** **as** **plt**

**from** **matplotlib.figure** **import** Figure

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

fig = plt.figure(figsize =(5, 4))

ax1 = fig.add\_axes([0.1, 0.1, 0.8, 0.8])

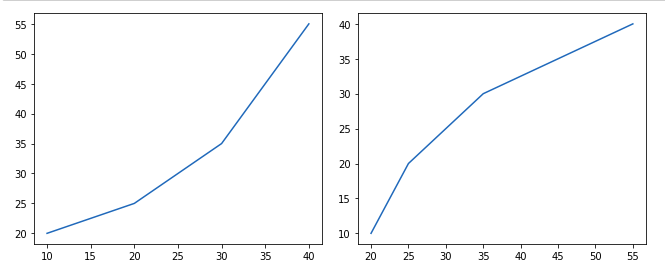
ax2 = fig.add\_axes([1, 0.1, 0.8, 0.8])

ax1.plot(x, y)

ax2.plot(y, x)

plt.show()

**Output:**



**Method 2: Using subplot()**

The [subplot()](https://www.geeksforgeeks.org/matplotlib-pyplot-subplot-function-in-python/) method adds a plot to a specified grid position within the current figure. It takes three arguments: the **number of rows, columns**and**the plot index.**

**Example:**

**import** **matplotlib.pyplot** **as** **plt**

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

plt.figure()

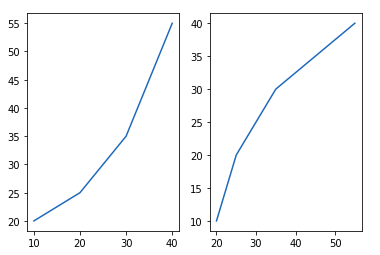
plt.subplot(121)

plt.plot(x, y)

plt.subplot(122)

plt.plot(y, x)

**Output:**



**Method 3: Using subplot2grid()**

The[subplot2grid()](https://www.geeksforgeeks.org/matplotlib-pyplot-subplot2grid-in-python/)creates axes object at a specified location inside a grid and also helps in spanning the axes object across multiple rows or columns.

**Example:**

**import** **matplotlib.pyplot** **as** **plt**

x = [10, 20, 30, 40]

y = [20, 25, 35, 55]

axes1 = plt.subplot2grid (

(7, 1), (0, 0), rowspan = 2, colspan = 1)

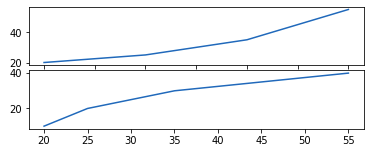
axes2 = plt.subplot2grid (

(7, 1), (2, 0), rowspan = 2, colspan = 1)

axes1.plot(x, y)

axes2.plot(y, x)

**Output:**



**Saving Plots Using savefig()**

When we create plots using Matplotlib sometimes we want to save them as image files so we can use them later in reports, presentations or share with others. Matplotlib provides the **savefig()** method to save our current plot to a file on our computer. We can [saving a plot](https://www.geeksforgeeks.org/how-to-save-a-plot-to-a-file-using-matplotlib/) in different formats like .png, .jpg, .pdf, .svg and more by just changing the file extension.

**Example:**

**import** **matplotlib.pyplot** **as** **plt**

year = ['2010', '2002', '2004', '2006', '2008']

production = [25, 15, 35, 30, 10]

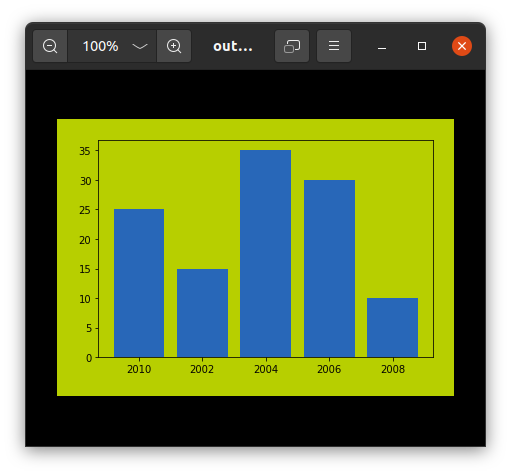
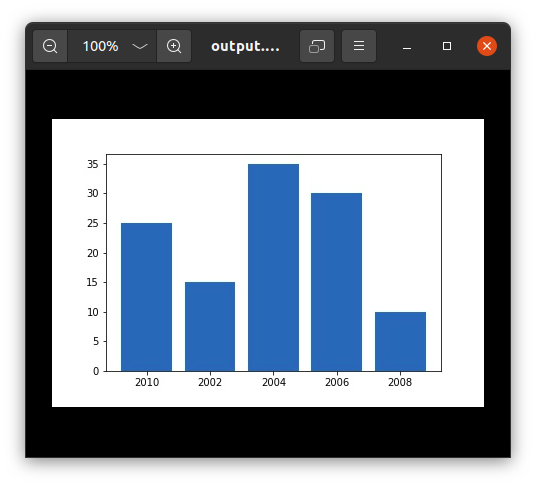
plt.bar(year, production)

plt.savefig("output.jpg")

plt.savefig("output1", facecolor='y', bbox\_inches="tight",

pad\_inches=0.3, transparent=**True**)

**Output:**



With these Matplotlib functions and techniques we can create clear, customized and insightful visualizations that bring our data to life.

s

**Objective:**

Create a box plot using matplotlib with sample data that includes:

* Normal distribution
* Skewed data
* Outliers
* Constant values
* Small sample size

We'll plot all of these in one chart and explain what each shows.

**✅ Code Example with Explanation:**

python

CopyEdit

import matplotlib.pyplot as plt

import numpy as np

# Set seed for reproducibility

np.random.seed(42)

# Sample datasets

normal\_data = np.random.normal(loc=50, scale=10, size=100) # normal distribution

skewed\_data = np.random.exponential(scale=20, size=100) # right-skewed

outlier\_data = np.append(np.random.normal(30, 5, 95), [100, 110, 120, 130, 150]) # some large outliers

constant\_data = np.full(100, 42) # same value repeated

small\_sample = np.random.normal(55, 2, size=5) # very small sample

# Combine data for plotting

data = [normal\_data, skewed\_data, outlier\_data, constant\_data, small\_sample]

# Box plot

plt.figure(figsize=(12, 6))

plt.boxplot(data, patch\_artist=True, labels=[

"Normal", "Skewed", "Outliers", "Constant", "Small Sample"

])

plt.title("Box Plot Examples for Data Preprocessing")

plt.ylabel("Value Range")

plt.grid(True)

plt.show()

**📊 Output Explanation**

**1. Normal Data**

* Symmetric box
* Median in center
* Whiskers are nearly equal
* No/few outliers

✅ Indicates clean, well-behaved data.

**2. Skewed Data**

* Longer upper whisker
* Median pushed downward
* Often indicates **log-normal** or **exponential** distributions

⚠️ Might need **log transformation** or **standardization**.

**3. Outlier Data**

* Many points shown as dots beyond whiskers
* Whiskers are short, box is small

⚠️ Consider **removing** or **treating outliers** via z-score, IQR filtering.

**4. Constant Data**

* Box is flat (height = 0)
* No whiskers, no spread

⚠️ Feature is **non-informative** → drop it during preprocessing.

**5. Small Sample**

* Box/whiskers may look odd or minimal
* Hard to assess distribution

⚠️ Small datasets may need **bootstrapping** or **augmentation**.

**✅ Summary**

| **Pattern** | **What It Suggests** | **Preprocessing Tips** |
| --- | --- | --- |
| Outliers | Extreme values affect models | Remove or transform them |
| Skewness | Distribution not symmetric | Try log, sqrt, or Box-Cox transformation |
| No variance | All values same | Drop that feature |
| Small samples | Too few data points | Augment or collect more data |