

# Data Visualization - Matplotlib

## Data Visualization

---

We, as humans, process visual information faster than text or numbers, so we use **Data visualization** i.e. the graphical representation of data to:

- Understand patterns and trends
- Identify outliers and anomalies
- Communicate insights clearly
- Support data-driven decision making

The common types of plots used in data visualization are line plots, bar charts, histograms, scatter plots, box plots etc.

## Matplotlib

---

**Matplotlib** is a **Python library** used for:

- Creating static, animated, and interactive plots
- Low-level control over plot appearance
- Serving as the foundation for other libraries (Seaborn, Pandas plotting)

## Installation

```
pip install matplotlib
```

## Usage

```
import matplotlib.pyplot as plt
```

## Basic Plot Structure

```
plt.plot(x, y)
plt.xlabel("X-axis label")
plt.ylabel("Y-axis label")
plt.title("Title")
plt.show()
```

## Important Plots in Matplotlib

### Line Plots

A line plot is a type of chart used to display data points connected by straight lines, typically to show trends or changes over a continuous variable, such as time or distance.

#### Example:

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

plt.plot(x, y)
plt.xlabel("X values")
plt.ylabel("Y values")
plt.title("Simple Line Plot")
plt.grid(True) # Adds a grid
plt.show()
```

Key features of a line plot:

- **X-axis:** usually continuous (time, sequence, measurements)
- **Y-axis:** values being measured
- **Line style:** solid (), dashed (-), dotted (:) etc.
- **Markers:** shows individual data points ( o, s, ^, \* , . )

## Styling a Line Plot

```
plt.plot(x, y,  
         color="blue",  
         linestyle="--",  
         linewidth=2,  
         marker="o",  
         markersize=6)  
  
plt.show()
```

## Multiple Lines on the Same Plot

```
y2 = [1, 3, 5, 7, 9]  
  
plt.plot(x, y, label="Line 1")  
plt.plot(x, y2, label="Line 2")  
  
plt.xlabel("X")  
plt.ylabel("Y")  
plt.legend()  
plt.show()
```

## Saving Plots

```
plt.savefig("line_plot.png")
```

## When to use?

Use a line plot when:

- Data points are ordered
- You want to show progression or trends
- You're comparing changes rather than individual categories

- 💡 In Matplotlib, `fmt` strings are a compact way to specify the line style, marker, and color in a single string, so `fmt` is a combination of `[color][marker][linestyle]`.

Example:

```
plt.plot(x, y, 'g--') # green dashed line
plt.plot(x, y, 'bo')   # blue circles, no line
```

Refer to documentation for all options -

[https://matplotlib.org/stable/api/\\_as\\_gen/matplotlib.pyplot.plot.html](https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html)

## Bar Charts

A bar chart is a graph that displays categorical data using rectangular bars. Each bar's height (or length, in horizontal bars) represents a value.

Example:

```
import matplotlib.pyplot as plt

x = ['A', 'B', 'C', 'D']
y = [10, 15, 7, 12]

plt.bar(x, y)
plt.xlabel("Categories")
plt.ylabel("Values")
plt.title("Simple Bar Chart")
plt.show()
```

This creates a vertical bar chart (most common).

## Styling a Bar Chart

```
plt.bar(x, y,
        color='skyblue',
        width=0.6,
        edgecolor='black',
        linewidth=1.5)

plt.text()
plt.show()
```

## Horizontal Bar Chart

```
plt.barh(x, y)
plt.xlabel("Values")
plt.ylabel("Categories")
plt.title("Horizontal Bar Chart")
plt.show()
```

## Multiple Bar Charts (Grouped Bars)

```
import numpy as np

x = np.arange(4)
y1 = [10, 15, 7, 12]
y2 = [8, 14, 9, 10]

width = 0.35

plt.bar(x - width/2, y1, width, label='Group 1')
plt.bar(x + width/2, y2, width, label='Group 2')

plt.xticks(x, ['A', 'B', 'C', 'D'])
plt.xlabel("Categories")
plt.ylabel("Values")
plt.legend()
plt.show()
```

## When to use?

Use a bar chart when:

- You are comparing **different categories**
- The data is **discrete**, not continuous
- You want clear visual comparison

💡 We can use `plt.text()` to add text annotations to the bars in the bar chart.

Syntax:

```
values = [10, 20, 30]
plt.bar(['A', 'B', 'C'], values)

for i, v in enumerate(values):
    plt.text(i, v, str(v))
```

## Scatter Plots

A scatter plot shows the relationship between two variables by plotting points on a 2D plane.

Example:

```
x = [1, 2, 3, 4, 5]
y = [2, 4, 1, 3, 5]

plt.scatter(x, y)
plt.xlabel("X values")
plt.ylabel("Y values")
plt.title("Simple Scatter Plot")
plt.grid(True)
plt.show()
```

## Styling a Bar Chart

```
plt.scatter(x, y,
            s=100,           # marker size
            c='red',          # color
            marker='o',        # marker shape
            alpha= 0.7)       # transparency

plt.show()
```

## Using a Colormap (Color by Value)

```
import numpy as np

colors = np.array([10, 20, 30, 40, 50])

plt.scatter(x, y, c=colors, cmap='viridis')
plt.colorbar(label='Color Scale')
plt.show()
```

## Multiple Scatter Plots

```
y2 = [5, 3, 4, 2, 1]

plt.scatter(x, y, label='Set 1')
plt.scatter(x, y2, label='Set 2')

plt.xlabel("X")
plt.ylabel("Y")
plt.legend()
plt.show()
```

## When to use?

Use a scatter plot when:

- You want to analyze the **relationship** between two variables
- Data is **numerical and continuous**
- You don't want to imply a trend by connecting points

 We can also add annotations. **Annotations** are used to add explanatory text or markers to specific points in a plot.

Example:

```
x = [1, 2, 3, 4, 5]
y = [5, 7, 6, 8, 7]

plt.scatter(x, y)

# Annotate each point
for i in range(len(x)):
    plt.text(x[i]+0.1, y[i]+0.1, f"({x[i]}, {y[i]})") # small offset
```

## Pie Charts

A pie chart is a circular chart divided into slices to show relative proportions of a whole.

- Each slice = a category
- Size of slice = value of that category relative to the total

We prefer them only when our data has **few categories**.

### Example:

```
sizes = [30, 20, 25, 25]
labels = ['A', 'B', 'C', 'D']

plt.pie(sizes, labels=labels)
plt.show()
```

### Adding Percentages

```
plt.pie(sizes,
         labels=labels,
         autopct='%1.1f%%') # shows percentage

plt.show()
```

### Changing Colors

```
colors = ['skyblue', 'lightgreen', 'pink', 'orange']

plt.pie(sizes, labels=labels, autopct='%1.1f%%', colors=colors)
plt.show()
```

### Starting Angle and Shadow

```
plt.pie(sizes,
         labels=labels,
         autopct='%1.1f%%',
         startangle=90, # rotate chart
         shadow=True) # adds shadow

plt.show()
```

## Wedgeprops

In Matplotlib pie charts, `wedgeprops` is a dictionary of properties that controls the appearance of the pie slices (wedges). It has common properties like `edgecolor`, `linewidth`, `linestyle`, `alpha` etc.

```
plt.pie(sizes,
         labels=labels,
         autopct='%1.1f%%',
         wedgeprops={'edgecolor': 'black', 'linewidth': 2})
```

## When to use?

Use a pie chart when:

- You want to display **part-to-whole relationships**
- You have **categorical data**
- Data has **few categories** (4–6); too many slices make it hard to read

## Histograms

A histogram shows the distribution of numerical data by:

- Dividing values into **bins** (ranges)
- Counting how many values fall into each bin (frequency)

## Example:

```
data = [1, 2, 2, 3, 3, 3, 4, 4, 5, 5, 5, 5]

plt.hist(data, bins=5, color='skyblue', edgecolor='black')

plt.xlabel("Value")
plt.ylabel("Frequency")
plt.title("Simple Histogram")
plt.grid(axis='y', linestyle='--', alpha=0.7)

plt.show()
```

## Number of bins

`bins` controls how the data is divided. It can be an integer or a sequence of bin edges:

```
plt.hist(data, bins=3) # integer  
  
plt.hist(data, bins=[1, 2, 3, 4, 5, 6]) # sequence
```

## Histogram Orientation

- Vertical (default)

```
plt.hist(data)
```

- Horizontal

```
plt.hist(data, orientation='horizontal')
```

## Multiple Histograms

```
data2 = [2, 3, 3, 4, 4, 5, 5, 5, 6]  
  
plt.hist(data, bins=5, alpha=0.5, label='Data 1', color='blue')  
plt.hist(data2, bins=5, alpha=0.5, label='Data 2', color='red')  
plt.legend()
```

`alpha` controls transparency so overlapping histograms are visible.

## When to use?

Use a histogram when:

- You want to show the **distribution of continuous data**.
- You have to analyze **frequency, spread, skewness**, or patterns.
- Working with **large datasets** to summarize trends.



`axvline` (Axis Vertical Line) is used to draw a **vertical line** at a specific x-value in a plot.

Example:

```
plt.axvline(x=value, color='color', linestyle='style', linewidth=width,  
label='label')
```

## Box Plots

A box plot is a statistical visualization that summarizes the distribution of a dataset using five key numbers:

- Minimum (lowest non-outlier)
- First Quartile (Q1) – 25th percentile
- Median (Q2) – 50th percentile
- Third Quartile (Q3) – 75th percentile
- Maximum (highest non-outlier)

It can also show **outliers**.

Example:

```
data = [7, 8, 5, 6, 9, 7, 8, 10, 4, 6]  
  
plt.boxplot(data)  
plt.ylabel("Values")  
plt.title("Simple Box Plot")  
plt.show()
```

## Main Values in the Box Plot

### 1. Minimum (Lower Whisker)

- The smallest data point within  $1.5 \times \text{IQR}$  below Q1
- Any smaller points are considered outliers

### 2. First Quartile (Q1)

- The 25th percentile of the data
- 25% of data points are below Q1
- Bottom edge of the box

### 3. Median (Q2)

- The 50th percentile (middle value)
- Line inside the box
- Splits the dataset into two halves

### 4. Third Quartile (Q3)

- The 75th percentile of the data
- 75% of data points are below Q3
- Top edge of the box

### 5. Maximum (Upper Whisker)

- The largest data point within  $1.5 \times \text{IQR}$  above Q3
- Points beyond this are outliers

### 6. Interquartile Range (IQR)

- Difference between Q3 and Q1:  $\text{IQR} = \text{Q3} - \text{Q1}$
- Represents the middle 50% of the data

### 7. Outliers

- Data points outside  $1.5 \times \text{IQR}$  from Q1 or Q3
- Plotted as dots or asterisks beyond the whiskers

## Horizontal Box Plot

```
plt.boxplot(data, vert=False)
plt.xlabel("Values")
plt.show()
```

## Multiple Box Plots

```
data1 = [7, 8, 5, 6, 9, 7, 8, 10, 4, 6]
data2 = [5, 6, 7, 8, 5, 4, 6, 7, 5, 6]

plt.boxplot([data1, data2], labels=['Dataset 1', 'Dataset 2'])
plt.show()
```

Each box represents a different dataset.

## Showing Mean

```
plt.boxplot(data, showmeans=True, meanline=True)
```

- `showmeans=True` adds the mean marker.
- `meanline=True` draws a line instead of a point.

## When to use?

- Visualize spread and skewness of data
- Identify outliers
- Compare distributions across multiple groups

## Stack Plots

A stack plot is a type of plot where multiple data series are stacked on top of each other. Each “layer” shows the contribution of one category, and the top line shows the cumulative total.

### Example:

```
x = [1, 2, 3, 4, 5]
y1 = [1, 2, 3, 4, 5]
y2 = [2, 1, 2, 1, 2]

plt.stackplot(x, y1, y2, labels=['Data1', 'Data2'], colors=['blue', 'green'])

plt.xlabel("X")
plt.ylabel("Y")
plt.title("Basic Stack Plot")
plt.legend(loc='upper left')

plt.show()
```

## When to use?

Use a stack plot when you want to:

- Show cumulative data over time
- Compare multiple components
- Highlight trends in parts and whole

## Subplots

In Matplotlib, subplots allow you to display multiple plots in a single figure, arranged in a grid. This is useful for comparing different datasets or visualizations side by side.

### Example:

```
x = [1, 2, 3, 4, 5]
y1 = [2, 3, 5, 7, 11]
y2 = [1, 4, 2, 5, 3]

# 1 row, 2 columns, first subplot
plt.subplot(1, 2, 1) # (rows, columns, index)
plt.plot(x, y1, color='blue', marker='o')
plt.title("First Subplot")
plt.xlabel("X")
plt.ylabel("Y1")

# 1 row, 2 columns, second subplot
plt.subplot(1, 2, 2)
plt.plot(x, y2, color='red', marker='s')
plt.title("Second Subplot")
plt.xlabel("X")
plt.ylabel("Y2")

plt.tight_layout()
plt.show()
```

- `plt.subplot(nrows, ncols, index)` selects the current axes.
- Index counts row-wise from top-left to bottom-right (Row major).

## Modern Matplotlib

Modern Matplotlib with the **Object-Oriented (OO)** approach is now the recommended way to create professional plots.

### Why Use OO Style?

- More control over multiple axes, subplots, and figures
- Cleaner and more readable for complex plots
- Avoids side effects of `plt` (state-based interface)
- Easier to combine multiple plot types

## Basic Plot

```
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]

# Create figure and axes
fig, ax= plt.subplots()

# Plot data
ax.plot(x, y, label="Prime numbers", color='blue', linestyle='--', marker='o')

# Add labels and title
ax.set_xlabel("X axis")
ax.set_ylabel("Y axis")
ax.set_title("OO Line Plot Example")

# Add legend and grid
ax.legend()
ax.grid(True)

plt.show()
```

`fig` and `ax` are core objects that give you full control over our plots.

- `fig` is Figure & it represents the entire figure or canvas.
- `ax` is Axes & it represents a single plot or graph within the figure.

## Subplots

```
fig, axs = plt.subplots(2, 2, figsize=(8, 6))

axs[0, 0].plot(x, y1)
axs[0, 0].set_title("Top Left")

axs[0, 1].bar(['A','B','C'], [3,5,2])
axs[0, 1].set_title("Top Right")

axs[1, 0].scatter(x, y2)
axs[1, 0].set_title("Bottom Left")

axs[1, 1].hist([1,2,2,3,3,3,4])
axs[1, 1].set_title("Bottom Right")

plt.tight_layout()
plt.show()
```

| *Keep Learning & Keep Exploring!*