Week 4: Data Visualization with Matplotlib

Introduction to Data Visualization

- Data visualization is a crucial aspect of the data analysis process.
- Visualizations help in understanding patterns, trends, and insights in data that may not be apparent in raw numbers.
- Effective communication of findings is enhanced through visual representations.

Data Visualization with Matplotlib

Why Matplotlib?

- Matplotlib is a popular Python library for creating static, animated, and interactive visualizations.
- Matplotlib is a powerful and widely used 2D plotting library in Python.
- Matplotlib is highly customizable, allowing users to tailor the appearance of plots to meet specific needs.
- Its integration with NumPy makes it suitable for visualizing data from various scientific and engineering disciplines.

Pyplot module in Matplotlib

- In Matplotlib, pyplot is a collection of functions that make Matplotlib work like MATLAB.
- It provides a convenient interface for creating various plots and charts.
- When you import matplotlib.pyplot, you are essentially importing this module, which is commonly used for creating static, animated, and interactive plots.
- Matplotlib provides the foundational tools for creating visualizations, while matplotlib.pyplot simplifies the creation of common plots through a convenient interface.
- Pyplot contains functions such as plot(), scatter(), show(), etc., that allow you to create various types of plots with minimal code.
- By convention, the pyplot module is often imported as **plt** to make the code more concise.

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

Most common plots in Data Visualization

1) Line plots
2) Scatter plots
3) Bar plots
4) Histograms
5) Pie chart
6) Box plots

Line Plots

- Line plots are one of the most common types of plots used in data analysis.

- They display data points connected by straight-line segments, making them suitable for visualizing trends, patterns, and relationships in sequential data.

- Line plots can be enhanced with labels, titles, grid lines, and other customizations.

plot() function

- The matplotlib.pyplot.plot() function is a versatile method for creating line plots in Matplotlib.
- It allows you to visualize data by connecting data points with lines.

Basic Syntax:

plt.plot(x, y, format_string, **kwargs)

- It allows customization through the format_string argument, and additional features such as labels, legends, and gridlines enhance the clarity of the visualization.

plot() function

x and y (Required):

x: The data for the x-axis.

y: The data for the y-axis.

format_string (Optional):

A string that specifies the color, marker, and line style. For example, 'ro--' means red color, round markers, and a dashed line.

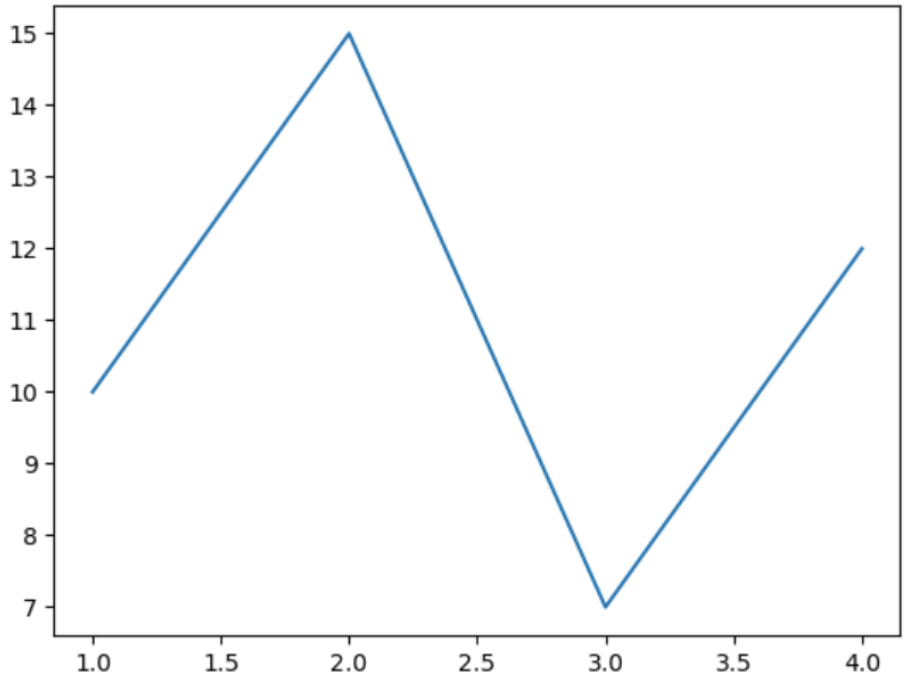
**kwargs (Optional):

Additional keyword arguments for customization.

Simple Line Plot

Example:

```
x = [1, 2, 3, 4]
y = [10, 15, 7, 12]

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



Common format_string Options:

Color:

- 'b': blue
- 'g': green
- 'r': red
- 'c': cyan
- 'm': magenta
- 'y': yellow
- 'k': black
- 'w': white

Marker:

- 'o': circle
- 's': square
- '^': triangle (up)
- 'v': triangle (down)
- '*': star
- '+': plus
- ',': pixel

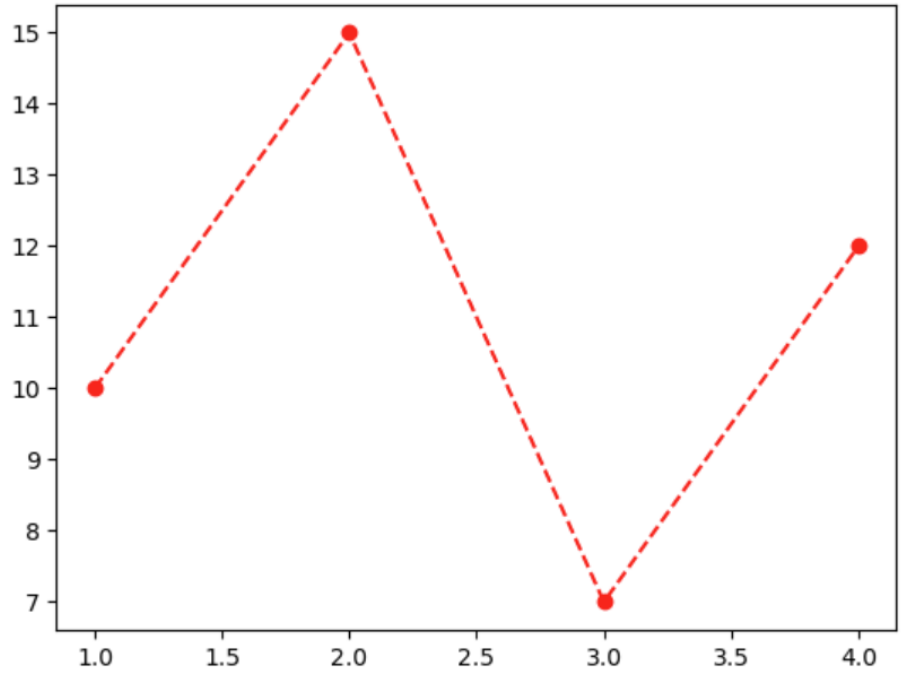
Line Style:

- '-': solid line
- ':': dotted line
- '--': dashed line
- '-.': dash-dot line

Line Plot with Formatting

Example:

```
x = [1, 2, 3, 4]
y = [10, 15, 7, 12]

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



Common Keyword Arguments (kwargs):

label: A label for the plot, which can be used in legends.

color: Sets the color of the line.

linestyle or ls: Determines the line style connecting the data points. (e.g., '-', '--', ':').

linewidth or lw: The width of the line.

marker: Specifies the marker style for data points (e.g., circles, squares).

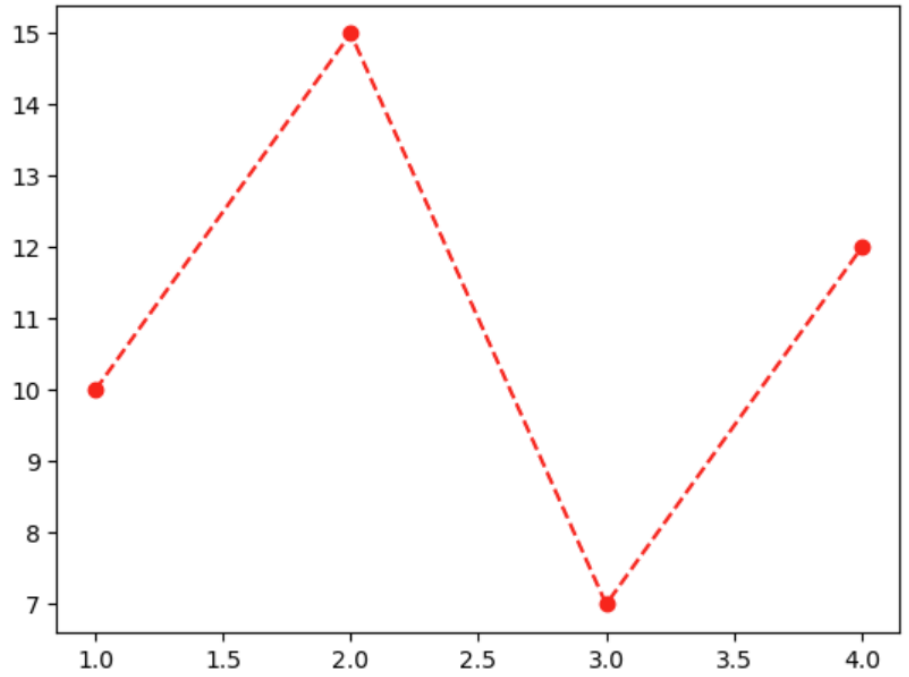
markersize or ms: The size of the markers.

- While the format string provides a quick way to specify basic visual elements, keyword arguments offer more detailed control over various aspects of the plot, allowing you to customize colors, line styles, markers, and more with precision. You can use them together for comprehensive customization.

Line Plot with keyword arguments

Example:

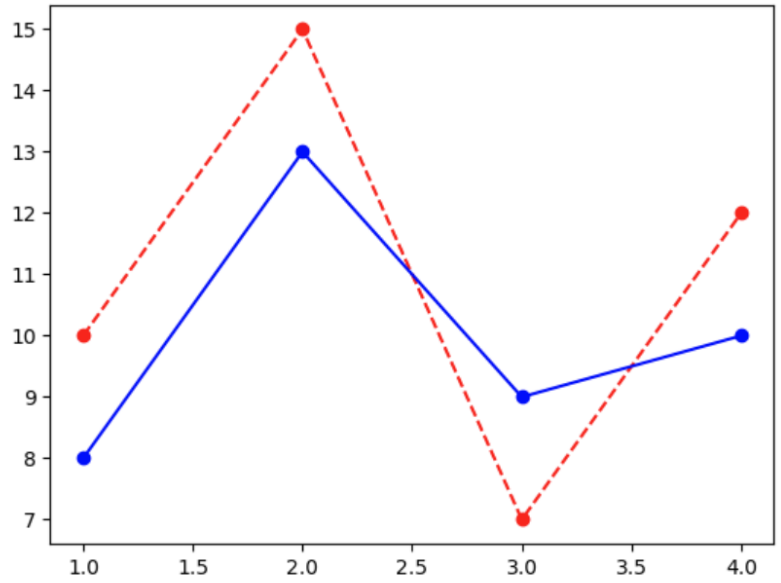
```
x = [1, 2, 3, 4]
y = [10, 15, 7, 12]

plt.plot(x, y, color = 'r', marker= 'o', linestyle = '--', label='Data Points')
plt.show()
```



Adding multiple lines in Line Plot

Example:

```
x = [1, 2, 3, 4]
y1 = [10, 15, 7, 12]
y2 = [8, 13, 9, 10]

# Plot the first line
plt.plot(x, y1, color = 'r', marker= 'o', linestyle = '--', label='First Line')

# Plot the second line
plt.plot(x, y2, color = 'b', marker= 'o', linestyle = '-', label='Second Line')

plt.show()
```



Additional Customization and Annotations

Matplotlib allows extensive customization to enhance the visual appeal of plots.

- Additional customization options like titles, legends, grid lines, and multiple lines.
- Adding annotations provides additional information directly on the plot.

Additional Customization

Labels and Title:

plt.xlabel('X-axis label')

plt.ylabel('Y-axis label')

plt.title('Plot Title')

Legend:

plt.legend(): Displays legend based on labels provided in plot().

Grid:

plt.grid(True): Displays grid lines.

Multiple Lines:

Call plot() multiple times before plt.show().

legend() function

- A legend provides information about the elements in the plot, making it easier for the reader to understand the significance of different lines or markers.
- The legend() function can take various parameters to customize the appearance and location of the legend.

Some common parameters include:

loc: Specifies the location of the legend on the plot. Common values include 'upper right', 'upper left', 'lower left', 'lower right', etc.

title: Sets the title of the legend.

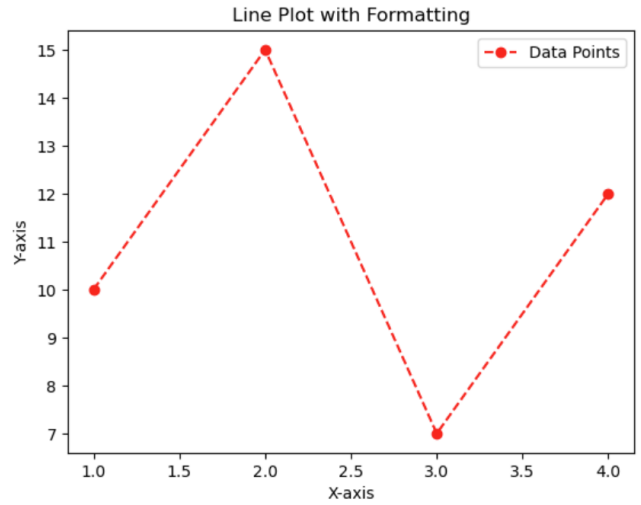
labels: Specifies the labels to use for each element in the legend.

bbox_to_anchor: Specifies the anchor point for the legend's bounding box. Useful for adjusting the position of the legend.

Line Plot with additional customization

- In this example, the legend() function is called after plotting the lines, and it automatically uses the labels provided in the plot() function.
- The legend is then displayed in the default location (usually the upper-right corner).

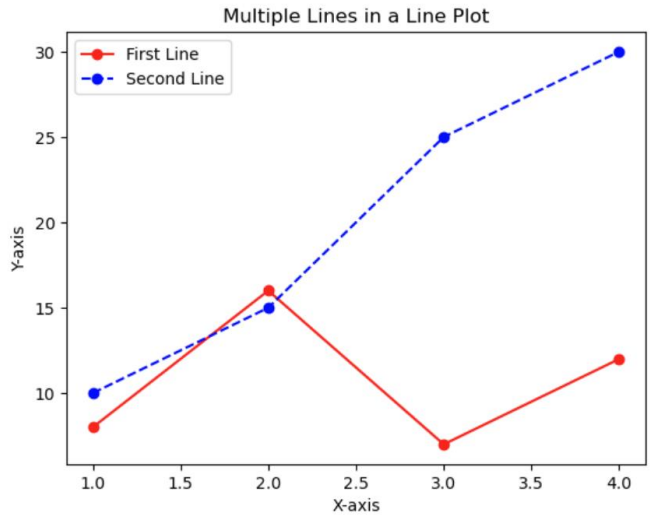
```
x = [1, 2, 3, 4]
y = [10, 15, 7, 12]

plt.plot(x, y, color = 'r', marker= 'o', linestyle = '--', label='Data Points')
plt.title('Line Plot with Formatting')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.legend()
plt.show()
```



Multiple lines with customization

Example:

```
x = [1, 2, 3, 4]
y1 = [8, 16, 7, 12]
y2 = [10, 15, 25, 30]

# Plot the first line
plt.plot(x, y1, color = 'r', marker= 'o', linestyle = '-', label='First Line')

# Plot the second line
plt.plot(x, y2, color = 'b', marker= 'o', linestyle = '--', label='Second Line')

# Set plot title and labels
plt.title('Multiple Lines in a Line Plot')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')

# Add legend
plt.legend()

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



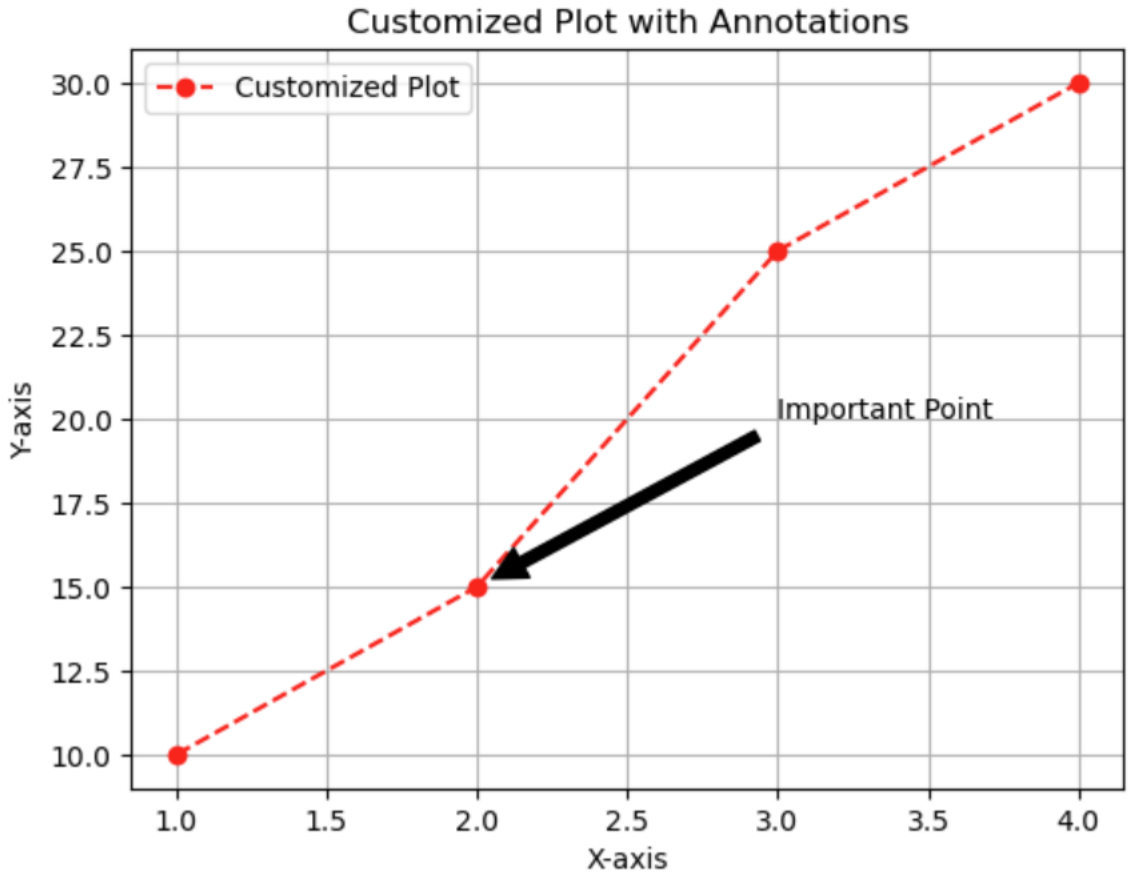
Annotations

Example:

```
x = [1, 2, 3, 4]
y = [10, 15, 25, 30]

plt.plot(x, y, color='r', marker='o', linestyle='--', label='Customized Plot')
plt.title('Customized Plot with Annotations')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.legend()

# Adding text annotation
plt.annotate('Important Point', xy=(2, 15), xytext=(3, 20), arrowprops=dict(facecolor='black', shrink=0.05))

# Show grid lines
plt.grid(True)

plt.show()
```

Annotations

Output:



Scatter Plots

- Scatter plots are valuable for displaying individual data points and exploring relationships between two continuous variables.
- Each point represents a single data observation, making it useful for identifying trends, clusters, or outliers in two-variable data.

Key Parameters:

marker: Specifies the marker style for data points (e.g., circles, squares).

color: Sets the color of the markers.

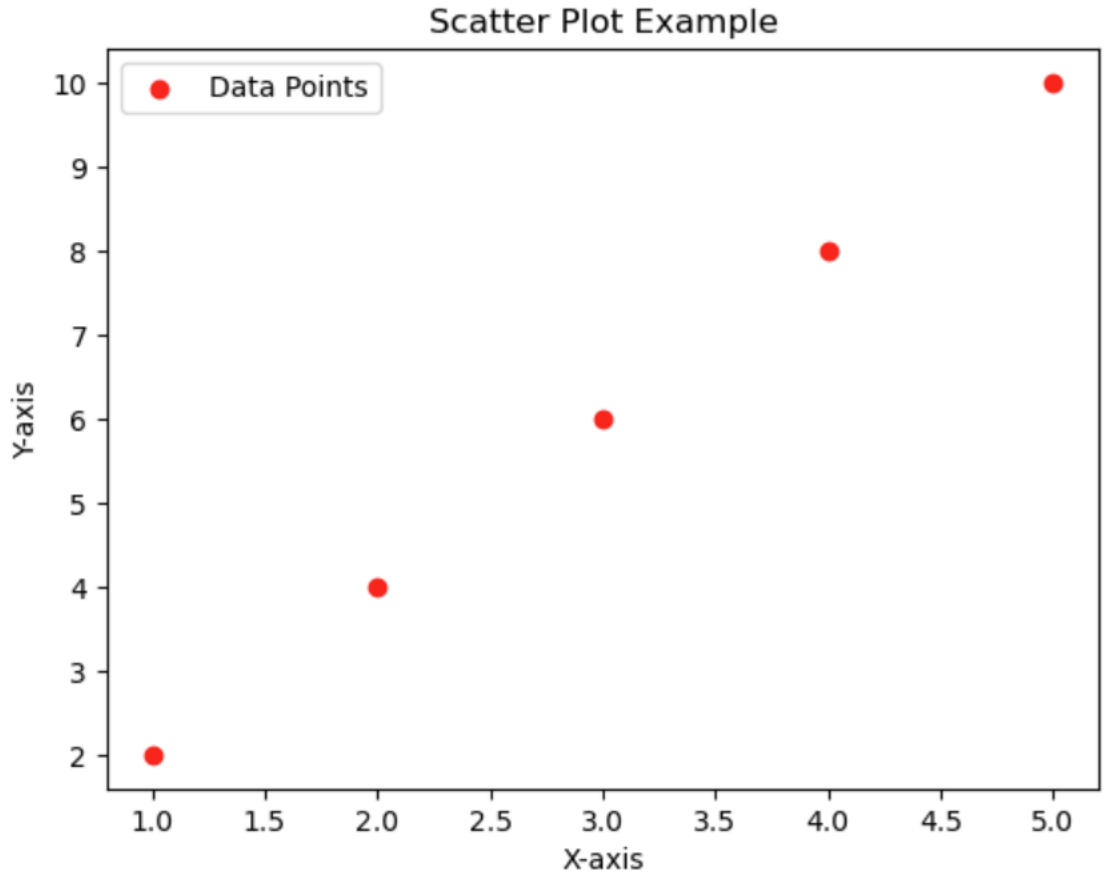
Scatter Plots

Example:

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

# Creating a Scatter Plot
plt.scatter(x, y, marker='o', color='r', label='Data Points')

# Adding Labels and Title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Scatter Plot Example')

# Adding Legend
plt.legend()

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

Scatter Plots

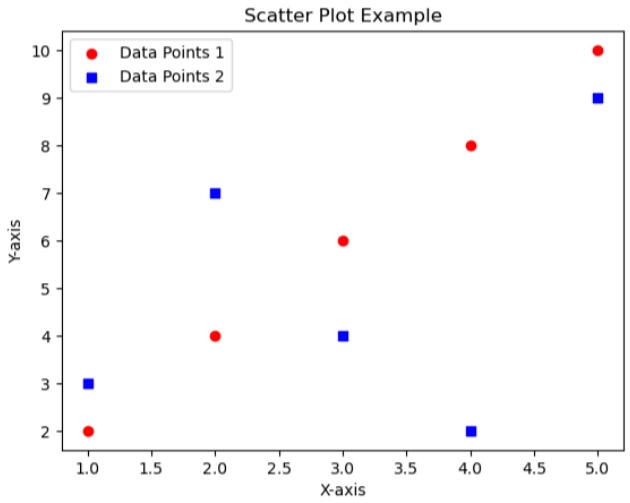
Output:



Multiple Scatter Plots

Example:

```
# Data
x = [1, 2, 3, 4, 5]
y1 = [2, 4, 6, 8, 10]
y2 = [3, 7, 4, 2, 9]

# Creating a Scatter Plot for datapoints 1
plt.scatter(x, y1, marker='o', color='r', label='Data Points 1')

# Creating a Scatter Plot for datapoints 1
plt.scatter(x, y2, marker='s', color='b', label='Data Points 2')

# Adding Labels and Title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Scatter Plot Example')

# Adding Legend
plt.legend()

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



Bar Charts

- Bar charts are widely used for comparing **values** across different **categories**.
- They represent data as rectangular bars with lengths proportional to the values they represent.

Key Parameters:

color: Sets the color of the bars.

alpha: Controls the transparency of the bars.

Tips: Consider horizontal bar charts (plt.barh()) for better readability in certain cases.

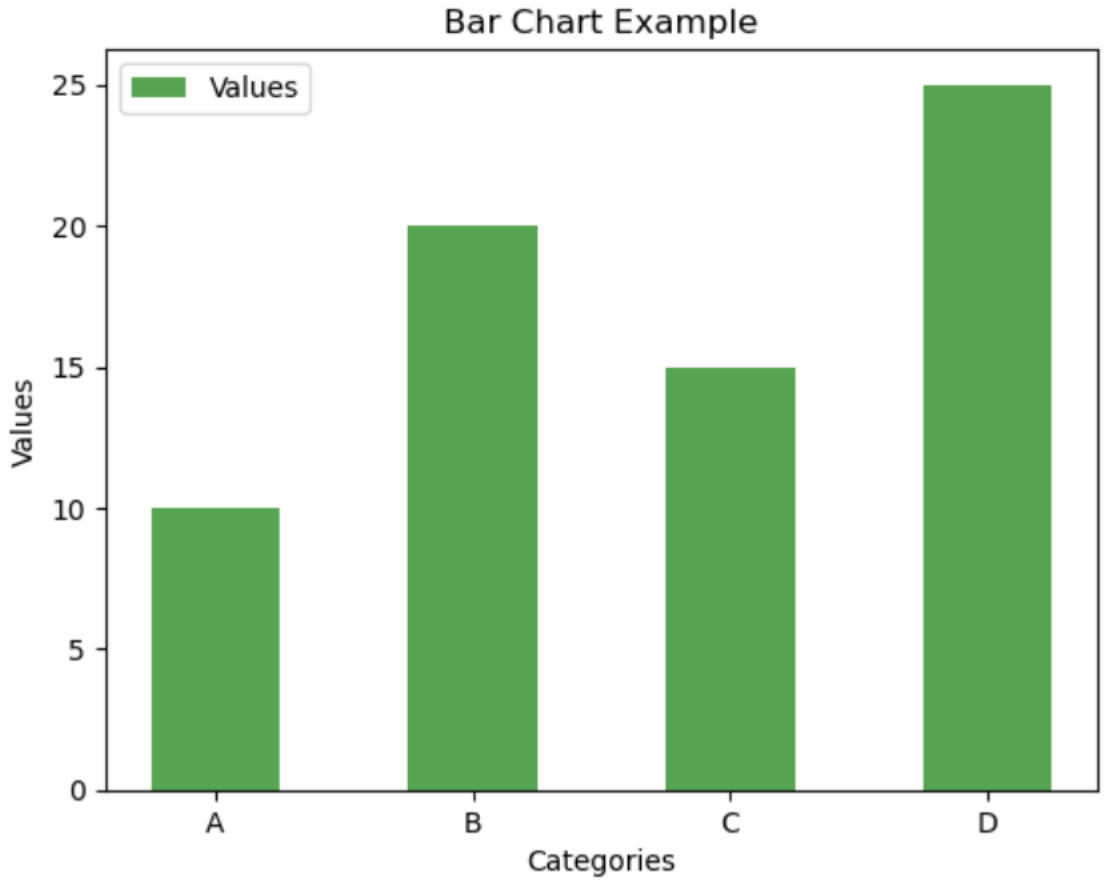
Bar Charts

Example:

```
# Data
categories = ['A', 'B', 'C', 'D']
values = [10, 20, 15, 25]

# Creating a Bar Chart
plt.bar(categories, values, color='green', alpha=0.7, width=0.5, label='Values')

# Adding Labels and Title
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Bar Chart Example')

# Adding Legend
plt.legend()

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

Bar Charts

Output:



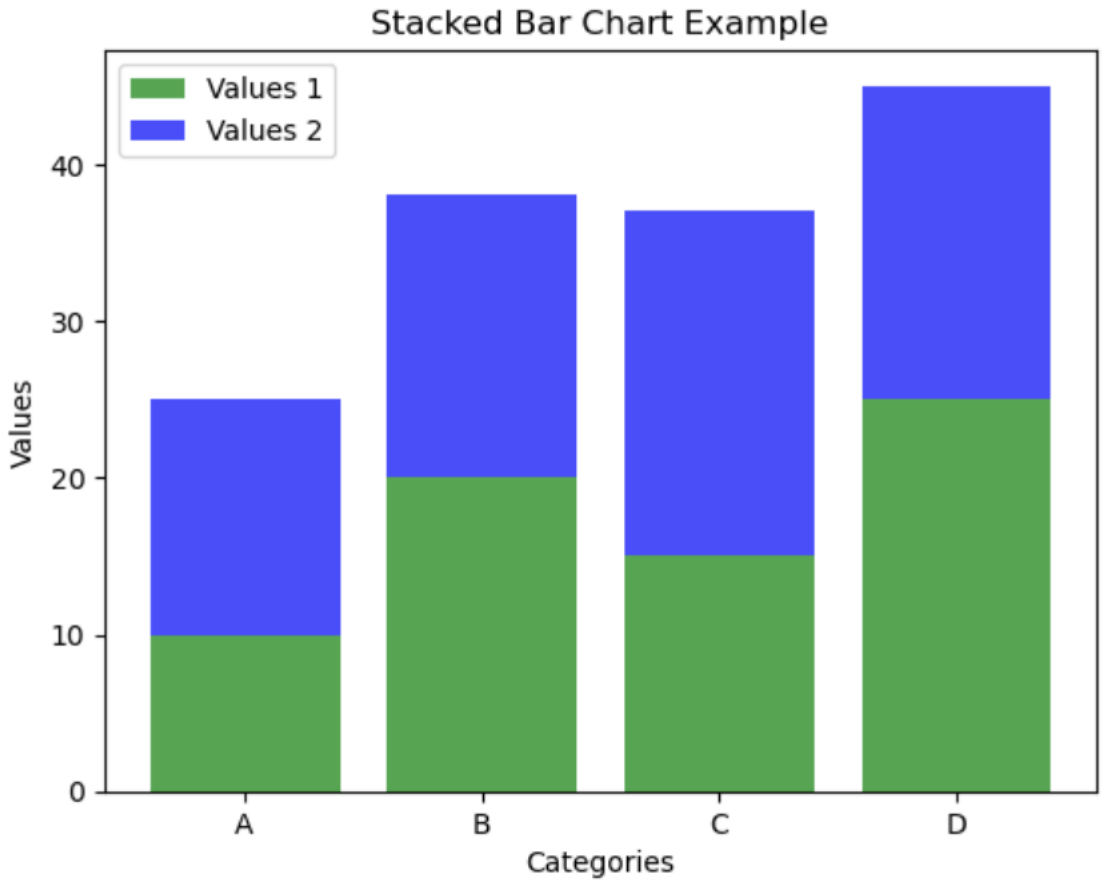
Stacked Bar Chart

Example:

```
# Data
categories = ['A', 'B', 'C', 'D']
values1 = [10, 20, 15, 25]
values2 = [15, 18, 22, 20]

# Creating a Stacked Bar Chart
plt.bar(categories, values1, color='green', alpha=0.7, label='Values 1')
plt.bar(categories, values2, bottom=values1, color='blue', alpha=0.7, label='Values 2')

# Adding Labels and Title
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Stacked Bar Chart Example')

# Adding Legend
plt.legend()

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

Stacked Bar Chart

Output:



Histograms

- Histograms visualize the distribution of a single continuous variable.
- They display the frequency of data points within specified bins.
- Effective for understanding the distribution and central tendency of a dataset.

Key Parameters:

bins: Specifies the number of bins or the bin edges.

color: Sets the color of the bars.

alpha: Controls the transparency of the bars.

edgecolor: Defines the color of the edges of the bars.

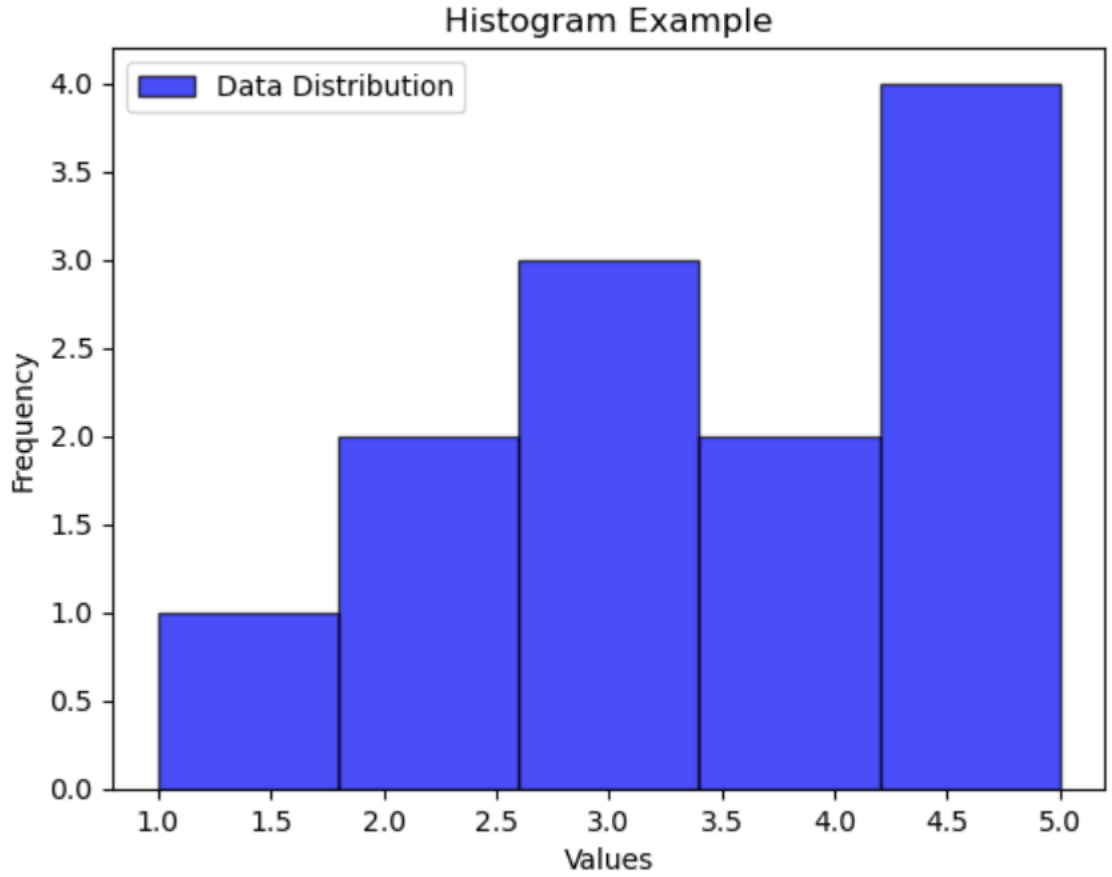
Histograms

Example:

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

# Creating a Histogram
plt.hist(data, bins=5, color='blue', alpha=0.7, edgecolor='black', label='Data Distribution')

# Adding Labels and Title
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.title('Histogram Example')

# Adding Legend
plt.legend()

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

Histograms

Output:



Pie Charts

- Pie charts represent the distribution of categorical data as slices of a circular chart.
- Each slice corresponds to a category, and the size of each slice indicates its proportion in the whole.
- Useful for showcasing proportions within a whole for categorical data.

Key Parameters:

labels: Specifies the labels for each category.

autopct: Displays the percentage distribution on each slice.

startangle: Rotates the pie chart to a specified angle.

Tip: Consider grouping small categories into an "Other" category for simplicity.

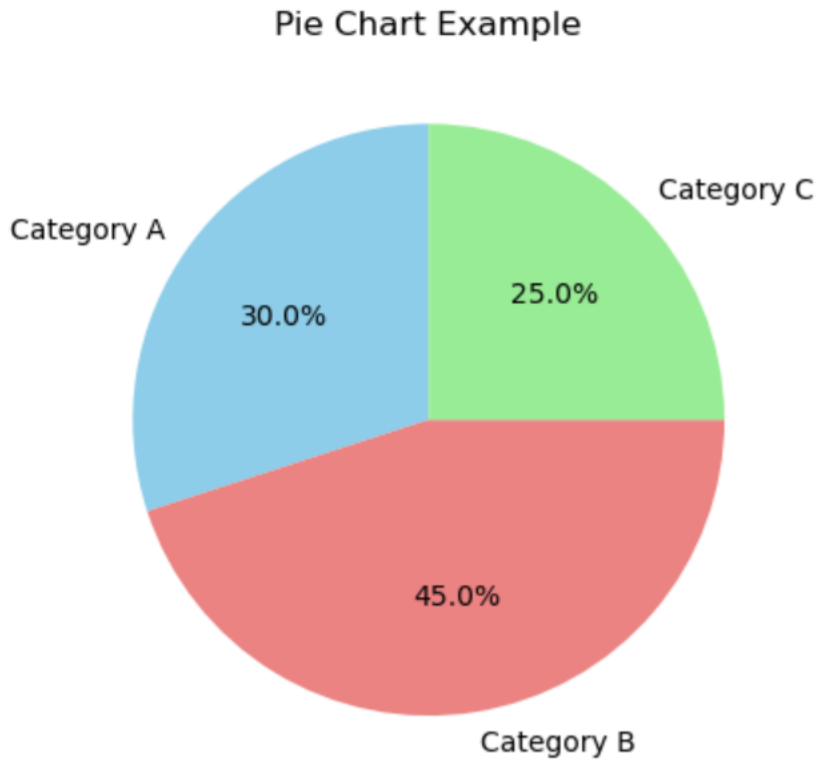
Pie Charts

Example:

```
# Data
labels = ['Category A', 'Category B', 'Category C']
sizes = [30, 45, 25]

# Creating a Pie Chart
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90, colors=['skyblue', 'lightcoral', 'lightgreen'])

# Adding Title
plt.title('Pie Chart Example')

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

Pie Charts

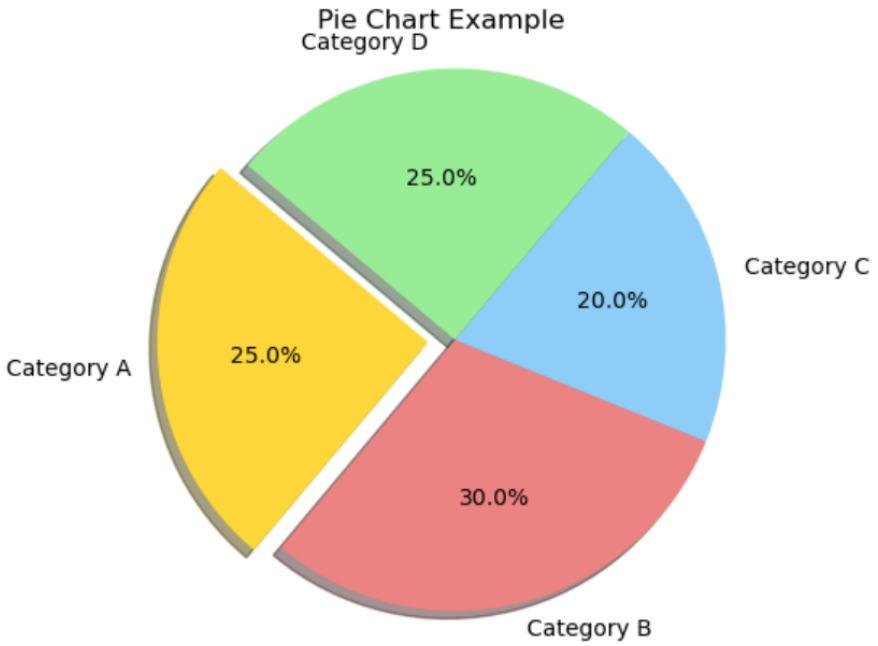
Output:



Example Pie Chart

Example:

```
labels = ['Category A', 'Category B', 'Category C', 'Category D']
sizes = [25, 30, 20, 25]
colors = ['gold', 'lightcoral', 'lightskyblue', 'lightgreen']
explode = (0.1, 0, 0, 0)  # explode first slice
plt.pie(sizes, explode=explode, labels=labels, colors=colors, autopct='%1.1f%%', shadow=True, startangle=140)
plt.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.title('Pie Chart Example')
plt.show()
```



Box Plots

- Box plots are useful for comparing distributions and identifying potential outliers in a dataset.
- Box plots, also known as box-and-whisker plots, depict the distribution and spread of numerical data.
- The components of a box plot, including the median, quartiles, and whiskers.

Key Components:

Box: Represents the interquartile range (IQR), covering the central 50% of the data.

Line inside the box: Represents the median (second quartile, Q2).

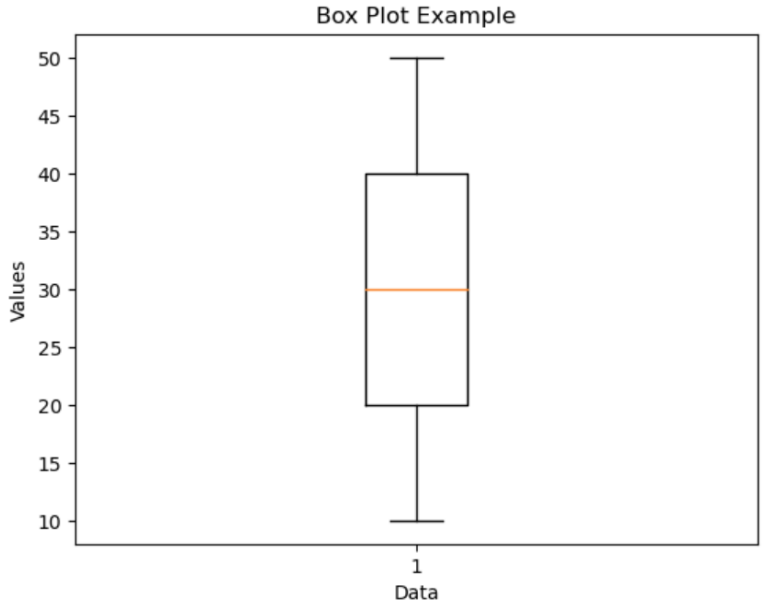
Whiskers: Indicate the range of the data, excluding potential outliers.

Outliers: Individual data points outside the whiskers.

Box Plots

Example:

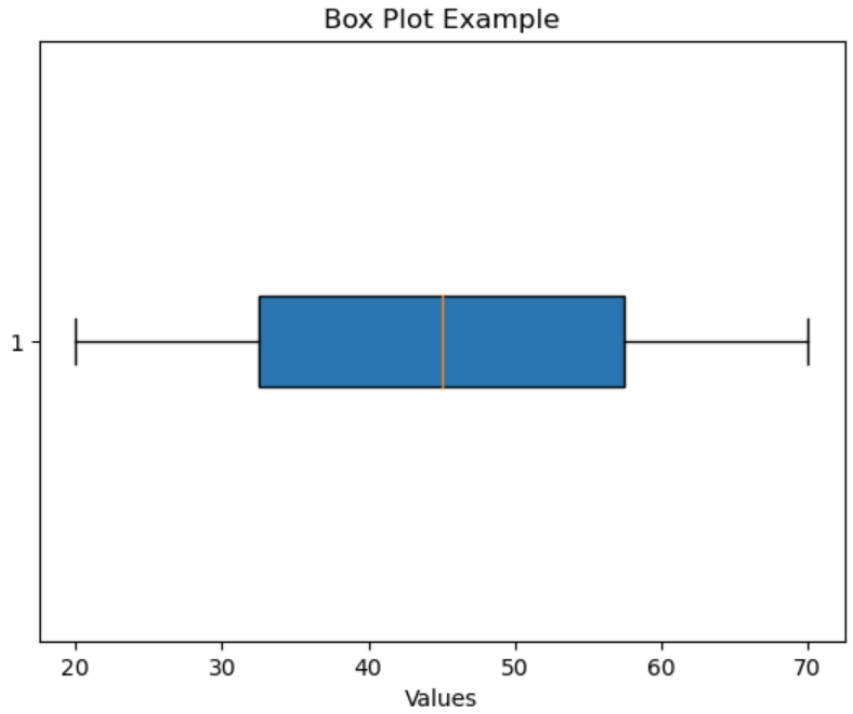
```
# Data
data = [10, 15, 20, 25, 30, 35, 40, 45, 50]

# Creating a Box Plot
plt.boxplot(data)

# Adding Title and Labels
plt.title('Box Plot Example')
plt.xlabel('Data')
plt.ylabel('Values')

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



Box Plots

Example:

```
data = [20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70]
plt.boxplot(data, vert=False, patch_artist=True)
plt.xlabel('Values')
plt.title('Box Plot Example')
plt.show()
```



Interquartile Range (IQR)

- The IQR is a measure of statistical dispersion, representing the range within which the middle 50% of the data points lie.
- It is calculated as the difference between the third quartile (Q3) and the first quartile (Q1):

$$IQR = Q3 - Q1$$

- The box in the boxplot represents the IQR. The lower and upper boundaries of the box correspond to Q1 and Q3, respectively.

Whiskers

- Whiskers in a boxplot represent the range of the data beyond the box.
- The length of the whiskers is often determined by a multiplier (default is 1.5 times the IQR) and extends to the data points within this range.
- Whiskers can provide an indication of the spread of the majority of the data.
- Outliers, which are data points beyond the whiskers, are sometimes plotted individually.

Subplots and multiple plots

- Subplots enable the creation of multiple plots within a single figure.
- Useful for comparing different visualizations or displaying related information.
- Visualize multiple aspects of a dataset side by side.

Key Concepts:

plt.subplot(rows, columns, index): Specifies the layout of subplots.

plt.figure(figsize=(width, height)): Adjusts the figure size.

plt.tight_layout(): Ensures proper spacing between subplots.

subplot() function

- The subplot() function in Matplotlib is used to create a grid of subplots within a single figure.
- This allows you to display multiple plots side by side or in a grid layout.

Syntax:

```
plt.subplot(nrows, ncols, index, **kwargs)
```

Parameters:

- nrows: Number of rows in the subplot grid.
- ncols: Number of columns in the subplot grid.
- index: Index of the subplot. Subplots are numbered from left to right, top to bottom, starting from 1.
- **kwargs: Additional keyword arguments for customization.

subplot() function

Figure:

- The Figure is the top-level container or window where the entire plot or visualization is drawn.
- It acts as a canvas that holds everything, including multiple subplots, axes, titles, legends, etc.
- When you create a plot in Matplotlib, you implicitly create a figure.

Axes:

- An Axes is the region of the figure where you can plot your data. It represents an individual plot or chart.
- A figure can have multiple axes arranged in a grid, creating subplots.
- Each subplot or individual plot is an instance of the Axes class.
- The Axes contains two (or three in the case of 3D plots) Axis objects representing the x and y (and z) axes.

subplot() function

- The syntax **fig, ax = plt.subplots()** is a common and convenient way to create a figure and a set of subplots (axes) at the same time.

Let's break down the components:

plt.subplots(): This function returns a tuple containing a figure (fig) and an array of axes (ax). If you don't specify the number of rows and columns for subplots, it creates a single subplot by default.

subplot() function

fig, ax = ...: This is tuple unpacking in Python. It allows you to assign the elements of a tuple to separate variables.

- In this case, **fig** is assigned to the figure, and **ax** is assigned to the array of axes.
- This syntax is concise and commonly used in Matplotlib because it provides a clean way to access both the figure and axes, making it easier to work with the plot's elements.
- You can also use it with multiple subplots, and the returned axes will be an array.

For instance, *fig, axs = plt.subplots(2, 2)* would create a 2x2 grid of subplots, and axs would be a 2D array of axes.

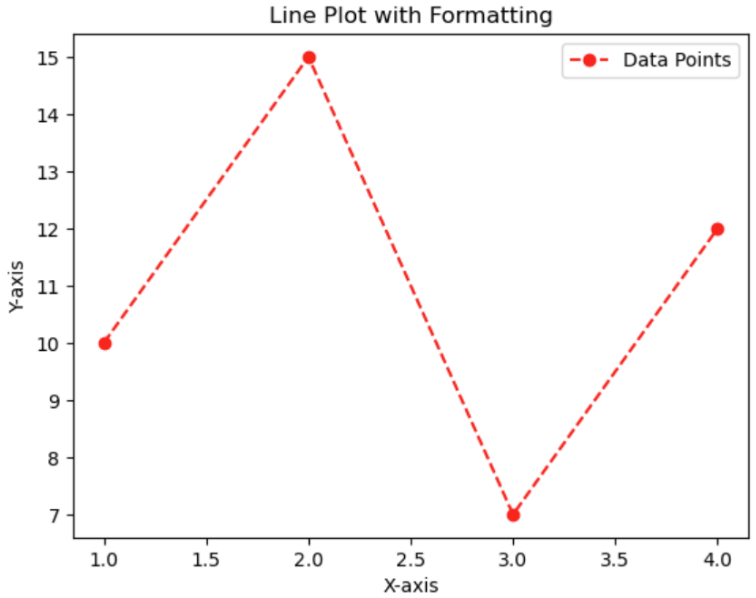
Single subplot

- In this example, plt.subplots() is used to create a figure (fig) and a single subplot (ax).
- The plot function is then used to plot data on the Axes (ax).
- Customizations like titles and labels are applied to the Axes.
- The plt.show() command is used to display the entire plot.

```
x = [1, 2, 3, 4]
y = [10, 15, 7, 12]

# Creating a Figure and Axes
fig, ax = plt.subplots()
# Now, 'fig' is the figure, and 'ax' is the single subplot (axes)

# You can plot on the axes
ax.plot(x, y, color = 'r', marker= 'o', linestyle = '--', label='Data Points')
ax.set_title('Line Plot with Formatting')
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.legend()
plt.show()
```

Single subplot

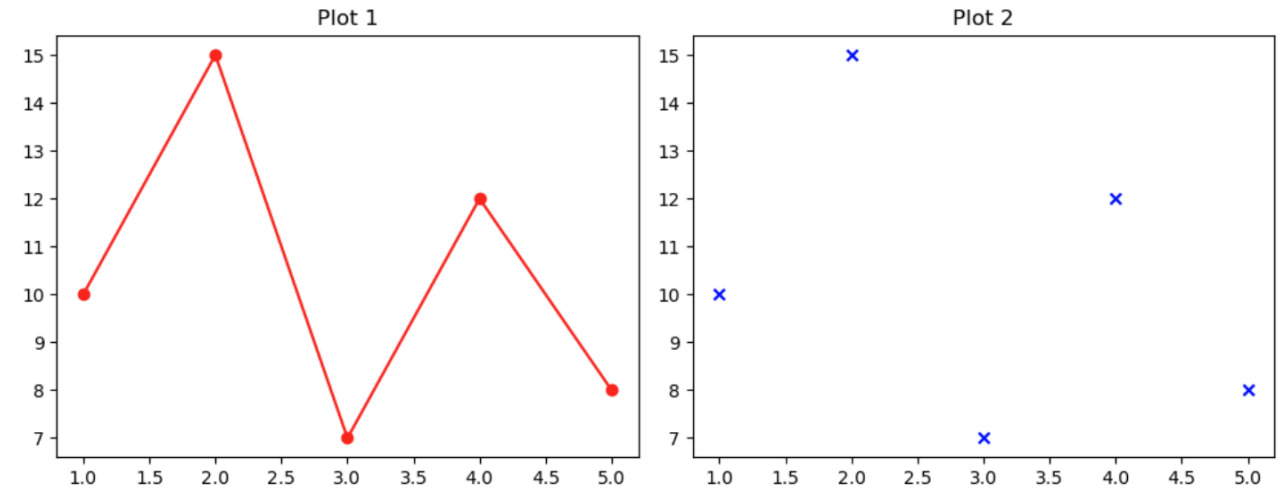
- plt.subplots(): Creates a new figure and returns a tuple containing the figure and a single Axes.
- ax.plot(): Plots data on the specified Axes.
- ax.set_xlabel(), ax.set_ylabel(): Set labels for the X and Y axes.
- ax.set_title(): Set the title of the plot.
- ax.legend(): Display a legend if multiple plots are present.



Multiple subplots

1x2 grid:

```
# Data
x = [1, 2, 3, 4, 5]
y = [10, 15, 7, 12, 8]

# Create a 1 x 2 grid of subplots
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))

# Plot 1
ax[0].plot(x, y, color='red', marker='o')
ax[0].set_title('Plot 1')

# Plot 2
ax[1].scatter(x, y, color='blue', marker='x')
ax[1].set_title('Plot 2')

plt.tight_layout()  # Adjust layout to prevent overlapping
plt.show()
```



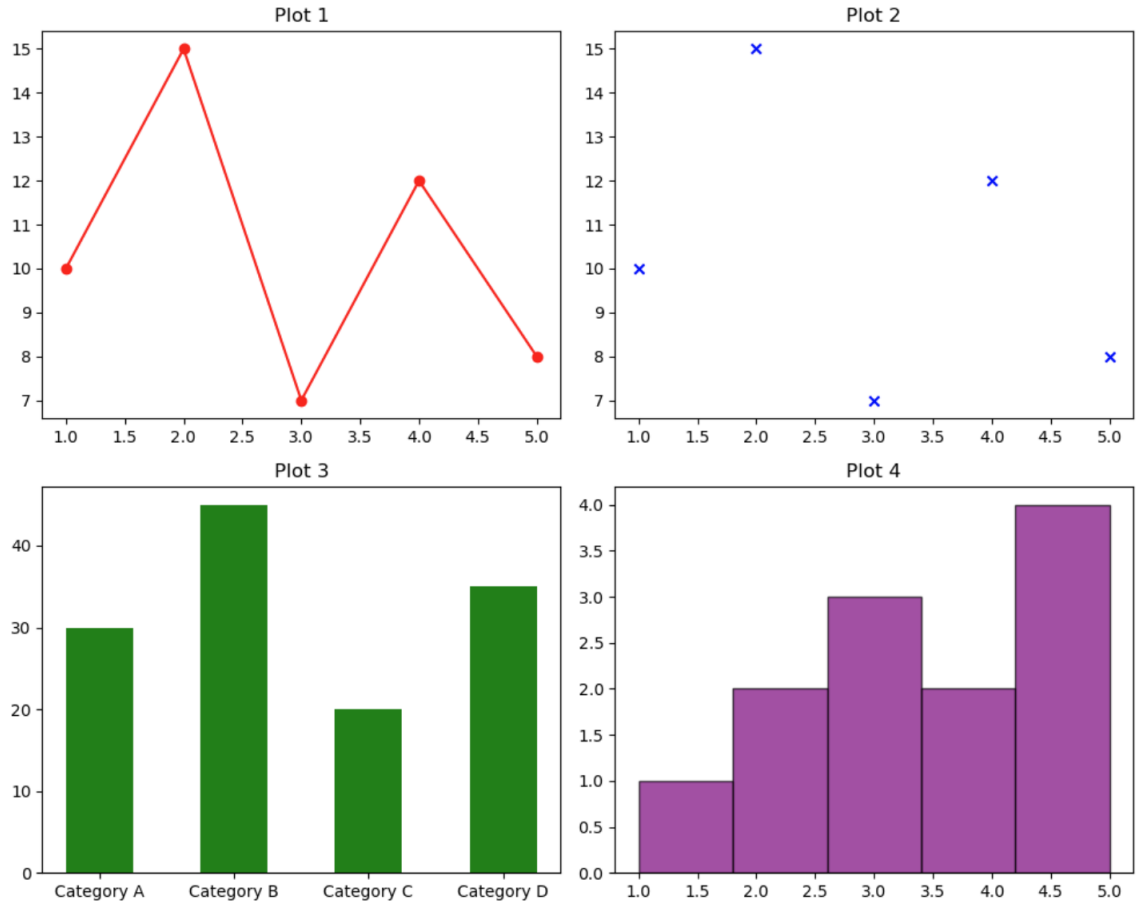
Multiple subplots

2x2 grid:

```
# Data we need for all the plots
x = [1, 2, 3, 4, 5]
y = [10, 15, 7, 12, 8]
categories = ['Category A', 'Category B', 'Category C', 'Category D']
values = [30, 45, 20, 35]
data = [1, 2, 2, 3, 3, 3, 4, 4, 5, 5, 5, 5]

# Create a 2 x 2 grid of subplots
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(10, 8))

# Plot 1
ax[0, 0].plot(x, y, color='red', marker='o')
ax[0, 0].set_title('Plot 1')

# Plot 2
ax[0, 1].scatter(x, y, color='blue', marker='x')
ax[0, 1].set_title('Plot 2')

# Plot 3
ax[1, 0].bar(categories, values, color='green', width=0.5)
ax[1, 0].set_title('Plot 3')

# Plot 4
ax[1, 1].hist(data, bins=5, color='purple', edgecolor='black', alpha=0.7)
ax[1, 1].set_title('Plot 4')

plt.tight_layout()  # Adjust layout to prevent overlapping
plt.show()
```

Multiple subplots

Output:



Saving and Exporting Plots

- Saving plots is crucial for sharing visualizations or incorporating them into reports and presentations or publications.
- Share visualizations in various formats on websites or social media.

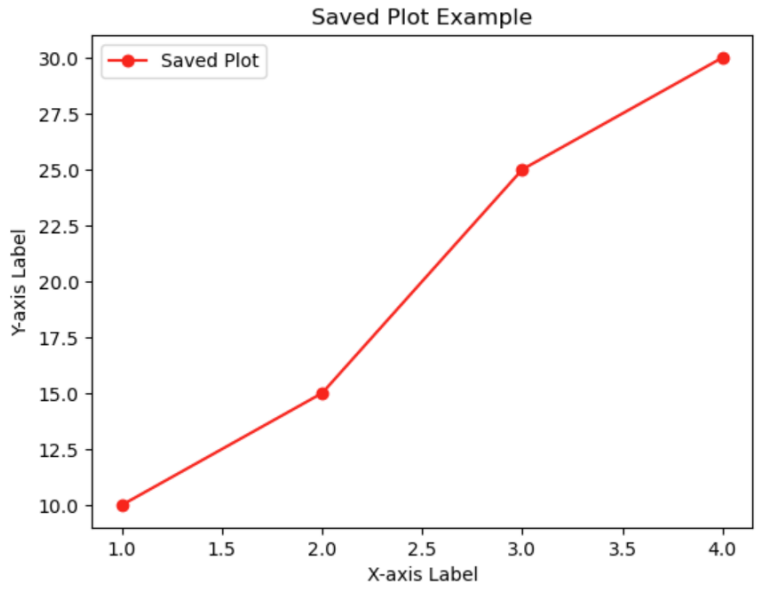
plt.savefig('filename.format', dpi=resolution)

- Saves the plot in the specified format and resolution.
- Supported formats include PNG, JPG, PDF, SVG, and more.

savefig() function

Example:

```
x = [1, 2, 3, 4]
y = [10, 15, 25, 30]

plt.plot(x, y, color='red', marker='o', linestyle='-', label='Saved Plot')
plt.xlabel('X-axis Label')
plt.ylabel('Y-axis Label')
plt.title('Saved Plot Example')
plt.legend()

# Saving the Plot to PNG format
plt.savefig('saved_plot.png', dpi=300)  # Specify file format and resolution

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



Advanced Data Visualization with Seaborn (EXTRA TOPIC)

- Seaborn builds on Matplotlib and provides a high-level interface for statistical graphics.
- It simplifies creating informative and attractive statistical graphics.
- Benefits of Seaborn:
  - Simplifies creating complex visualizations.
  - Enhances default Matplotlib plots with appealing styles. Seaborn's default color palettes enhance plot aesthetics.

1) Box plots
2) Heatmaps

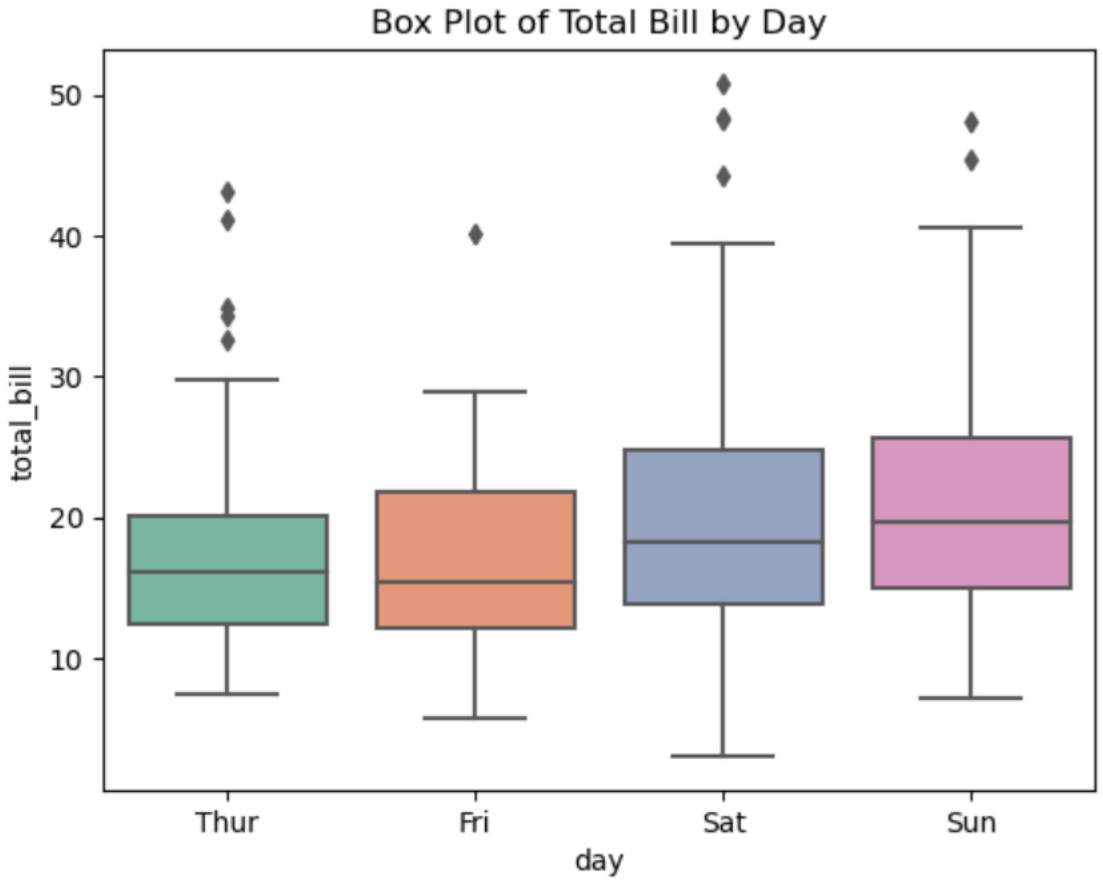
Box plots

sns.boxplot(x='variable', y='value', data=df, palette='color_palette')

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load Sample Dataset
tips = sns.load_dataset('tips')

# Box Plot
sns.boxplot(x='day', y='total_bill', data=tips, palette='Set2')
plt.title('Box Plot of Total Bill by Day')
plt.show()
```

Box plots

Output:



Heatmaps

- Heatmaps are effective for visualizing the correlation between variables in a dataset.
- Particularly useful for showcasing relationships between two categorical variables.
- Heatmaps visualize data in a matrix format, using color gradients to represent the intensity of values.

Key Parameters:

cmap: Specifies the color map for the heatmap.

interpolation: Defines the method for interpolating between data points.

Heatmap

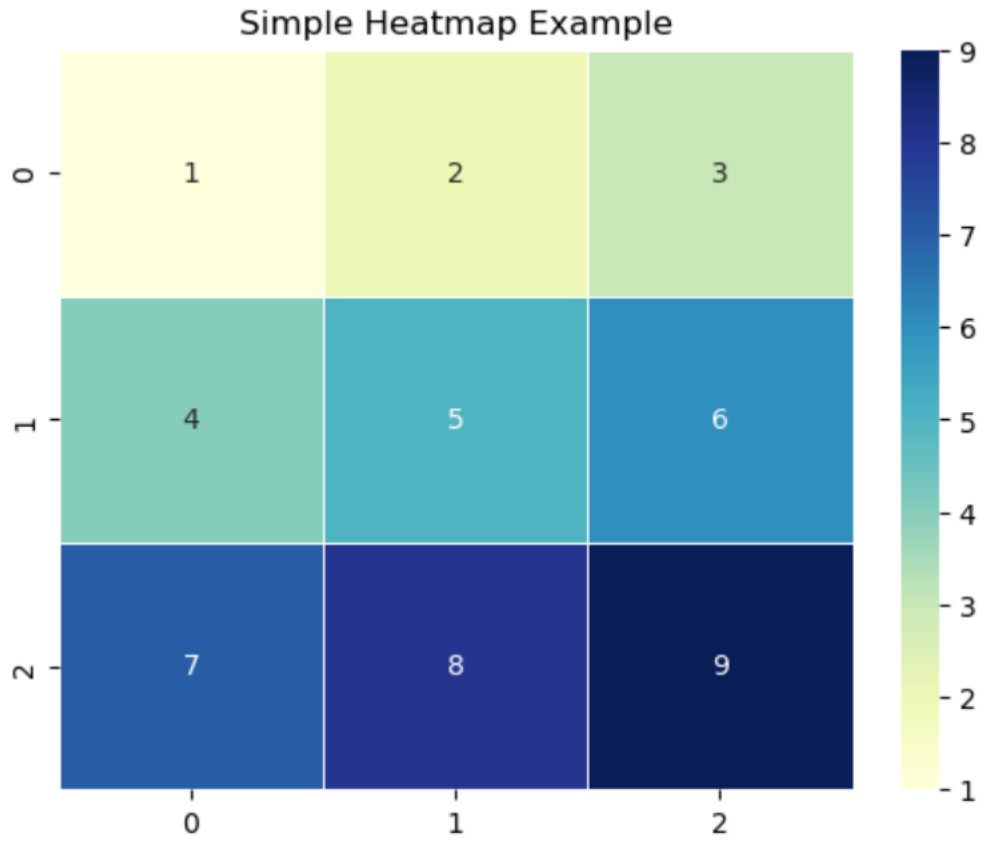
sns.heatmap(data, annot=True, cmap='color_map'):

Creates a heatmap of the correlation matrix.

```
import seaborn as sns
import matplotlib.pyplot as plt

# Sample Data
data = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Create DataFrame
df = sns.heatmap(data, cmap='YlGnBu', annot=True, fmt='d', linewidths=.5)

plt.title('Simple Heatmap Example')
plt.show()
```

Heatmap

Output:



Exercises

- Load the dataset called ‘titanic_clean.csv’ which is in Dataset folder on Microsoft Teams.

- ❑ Create a pie chart to visualize the distribution of passengers based on the 'Pclass' (Passenger Class) column.

- ❑ Create a box plot to display the distribution of ages ('Age') for different passenger classes ('Pclass’).

- ❑ Create a histogram to show the distribution of fares ('Fare') for the passengers.

Exercises

- Create a bar chart to compare the number of male and female passengers ('Sex').
- Create a scatter plot to explore the relationship between 'Age' and 'Fare'.
- Create a scatter plot to display the relationship between age and fare for passengers who survived (Survived = 1) and those who did not survive (Survived = 0). Use different colors for each group.