SUMMARY: MATPLOTLIB AND SEABORN

SESSION OVERVIEW:

In this session, the students will be able to:

- Understand the concept of data visualization
- Undertstand the importance of Matplotlib in data visualization
- Understand how to plot using Matplotlib
- Understand the integration of Matplotlib with NumPy
- Understand the integration of Matplotlib with Pandas
- Understand the importance of Seaborn in data visualization
- Understand common plot types in Seaborn

KEY POINTS AND EXAMPLES:

Understand the concept of data visualization

Data visualization refers to the graphical or pictorial representation of data using visual elements like graphs, charts, maps, etc. The purpose of plotting data is to visualize variations or show relationships between variables, making complex data more accessible, understandable, and usable.

We will learn how to visualise data using Matplotlib and Seaborn library of Python by plotting charts such as line, bar, scatter with respect to the various types of data.

Let's Embark with Matplotlib…

Understand the importance of Matplotlib in data visualization

Matplotlib is a powerful and versatile plotting library for Python that enables us to create static, animated, and interactive visualizations.

Matplotlib supports various types of plots including line plots, scatter plots, bar charts, histograms, pie charts, and more. This diversity allows us to represent data in different formats depending on the requirements.

It provides extensive options for customizing plots, such as adjusting colors, line styles, markers, fonts, annotations, and more. This level of customization helps in creating publication-quality plots tailored to specific needs.

Matplotlib integrates seamlessly with NumPy arrays and Pandas DataFrames, making it convenient to visualize data stored in these formats.

Installing Matplotlib

Installing Matplotlib is very similar to installing NumPy and Pandas. To install Matplotlib, you simply need to type:

```
conda install matplotlib
```

or

```
pip install matplotlib
```

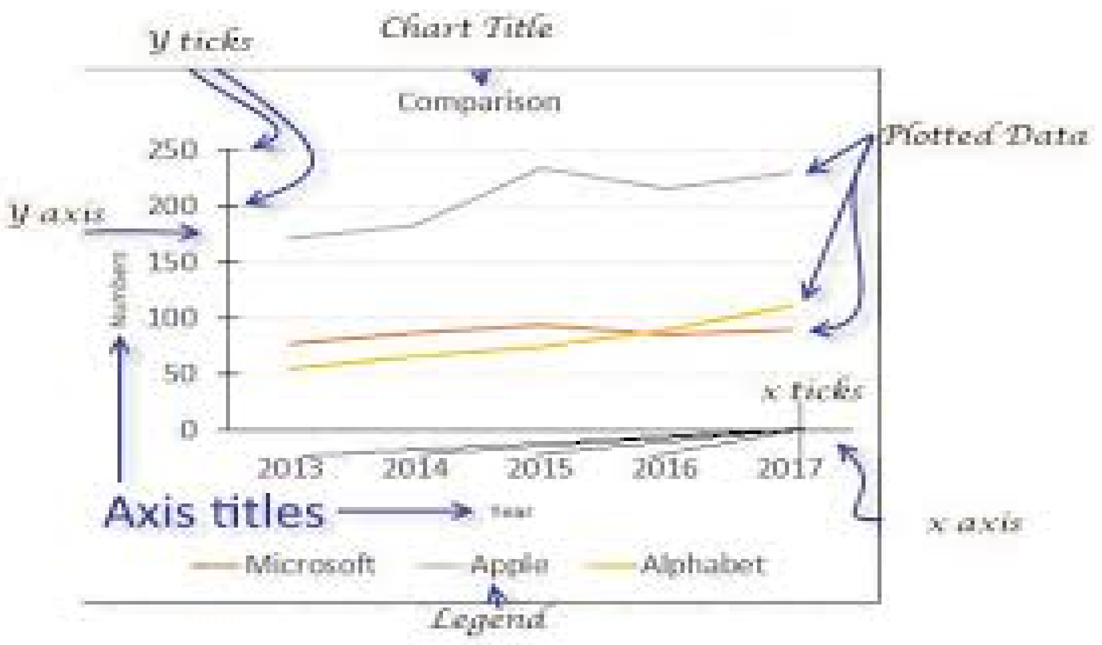
Importing Matplotlib

For plotting using Matplotlib, we need to import its Pyplot module using the following command:

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

After importing Matplotlib this way, you can use plt to access Matplotlib's plotting functions and create various types of plots such as line plots, scatter plots, bar charts, histograms, and more. This submodule (pyplot) provides a MATLAB-like interface for plotting, making it intuitive and easy to use for creating basic and advanced visualizations in Python.

Understand how to plot using Matplotlib



The pyplot module of Matplotlib contains a collection of functions that can be used to work on a plot. The plot() function of the pyplot module is used to create a figure. A figure is the overall window where the outputs of pyplot functions are plotted. A figure contains a plotting area, legend, axis labels, ticks, title, etc.

Key Components of a Figure:

- Plotting Area: The main area where data is visualized.
- Legend: A guide to the symbols, colors, and line styles used in the plot.
- Axis Labels: Labels for the x-axis and y-axis, describing the data.
- Ticks: Marks along the axes to indicate intervals or specific points.
- Title: The main heading of the plot, describing what the plot represents.

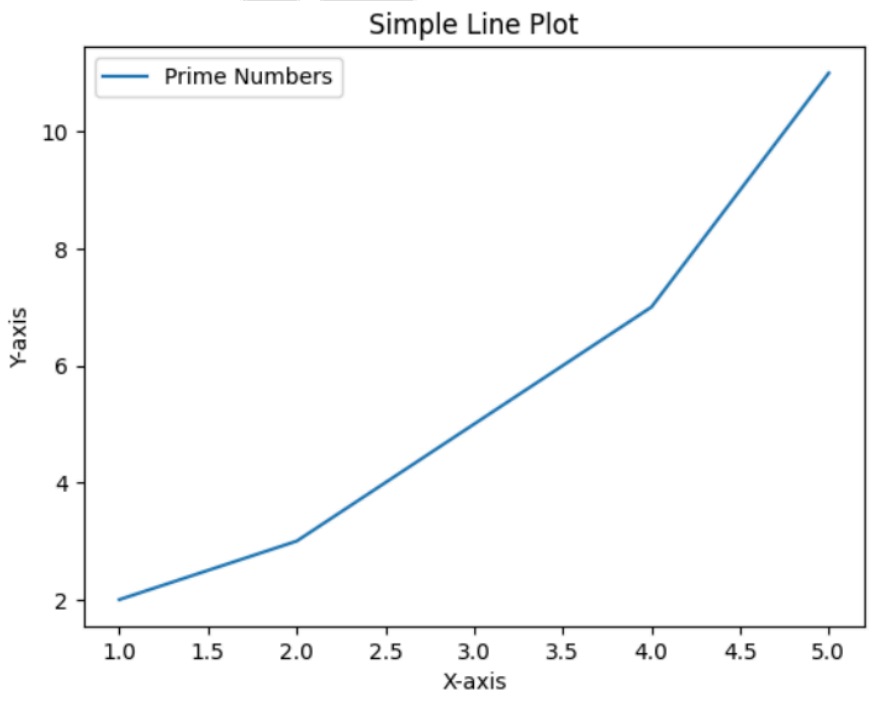
It is always expected that the data presented through charts easily understood. Hence, while presenting data we should always give a chart title, label the axis of the chart and provide legend in case we have more than one plotted data.

To plot x versus y, we can write plt.plot(x,y). The show() function is used to display the figure created using the plot() function.

```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]

# Creating a figure and a plot
```

```
plt.plot(x, y, label='Prime Numbers')

# Adding title and labels
plt.title('Simple Line Plot')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')

# Adding a legend
plt.legend()

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

The output will be:



In this example,

- Creating a Plot: The plt.plot(x, y) function creates a line plot using the data points in x and y.
- Adding Title and Labels: The plt.title('Simple Line Plot'), plt.xlabel('X-axis'), and plt.ylabel('Y-axis') functions add a title and labels to the x-axis and y-axis, respectively.
- Adding a Legend: The plt.legend() function adds a legend to the plot, explaining the data being represented.
- Displaying the Plot: The plt.show() function displays the plot in a new window.

This example demonstrates the basic structure and components of a figure in Matplotlib, showcasing the functionality provided by the pyplot module.

Marker

We can make certain other changes to plots by passing various parameters to the plot() function. It is also possible to specify each point in the line through a marker.

A marker is any symbol that represents a data value in a line chart or a scatter plot.

Below table shows a list of markers along with their corresponding symbol and description. These markers can be used in program codes:

Symbol	Symbol Description	Symbol	Symbol Description
.	Point	8	Octagon
,	Pixel	s	Square
o	Circle	p	Pentagon
v	Triangle Down	P	Plus (Filled)
^	Triangle Up	*	Star
<	Triangle Left	h	Hexagon1
>	Triangle Right	H	Hexagon2
1	Tri Down	+	Plus
2	Tri Up	x	X
3	Tri Left	X	X (Filled)
4	Tri Right	D	Diamond

Color

It is also possible to format the plot further by changing the color of the plotted data.

Below table shows the list of colours that are supported. We can either use character codes or the color names as values to the parameter color in the plot().

These are the predefined color abbreviations you can use for plotting in Matplotlib:

Character	Color
'b'	Blue
'g'	Green
'r'	Red
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

We can use standard color names directly as strings. Matplotlib supports a variety of color names such as 'blue', 'green', 'red', 'cyan', 'magenta', 'yellow', 'black', 'white', and more.

Hexadecimal color codes are another way to specify colors in Matplotlib. Hexadecimal color codes provide a wide range of colors with more precision and flexibility compared to named colors or abbreviations.

Hexadecimal Color Code Examples:

- '#FF5733': Represents a shade of orange.
- '#33FF57': Represents a shade of green.
- '#3357FF': Represents a shade of blue.

Linewidth and Line Style

The linewidth and linestyle property can be used to change the width and the style of the line chart.

Linewidth is specified in pixels. The default line width is 1 pixel showing a thin line. Thus, a number greater than 1 will output a thicker line depending on the value provided.

We can also set the line style of a line chart using the linestyle parameter. It can take a string such as "solid", "dotted", "dashed" or "dashdot".

Example: Consider the average heights and weights of persons aged 8 to 16 stored in the following two lists:

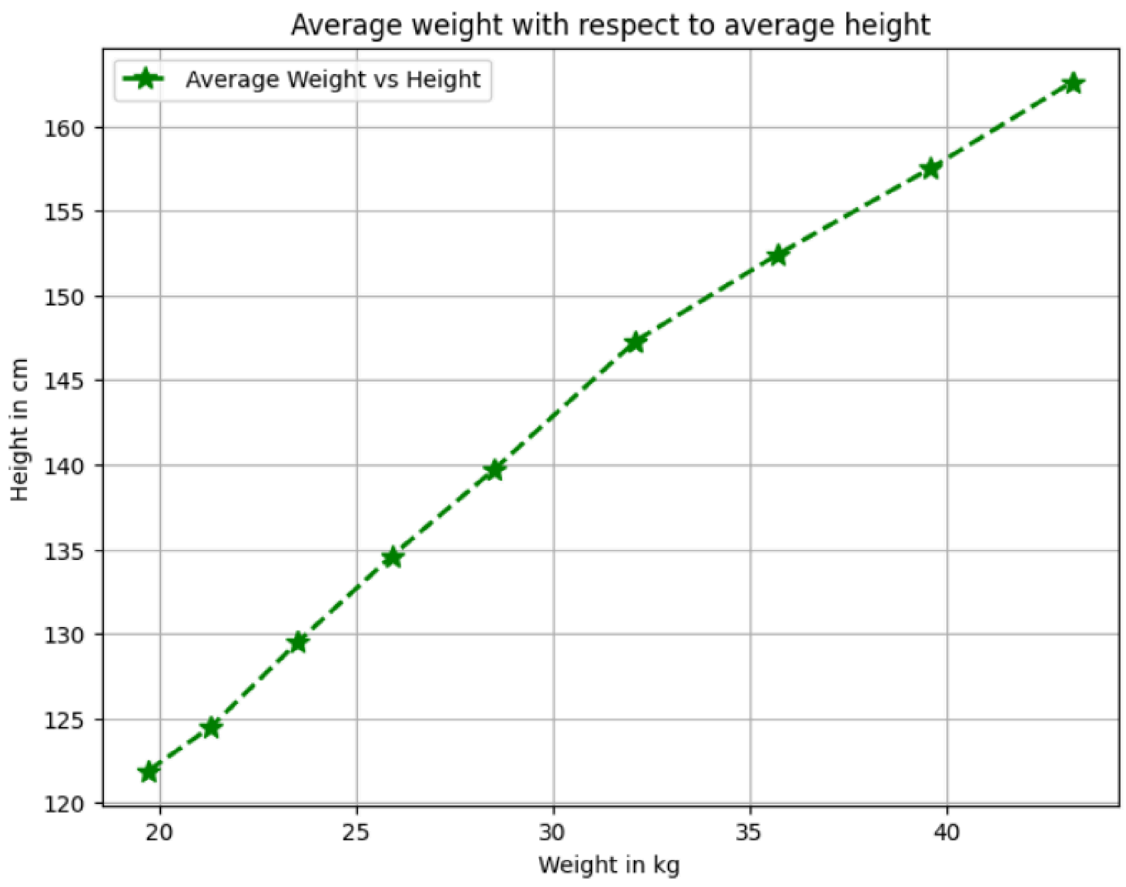
height = [121.9,124.5,129.5,134.6,139.7,147.3, 152.4, 157.5,162.6]
weight= [19.7,21.3,23.5,25.9,28.5,32.1,35.7,39.6, 43.2]

Plot a line chart where:

- x axis will represent weight
- y axis will represent height
- x axis label should be “Weight in kg”
- y axis label should be “Height in cm”
- colour of the line should be green
- use * as marker
- Marker size as10
- The title of the chart should be “Average weight with respect to average height”.
- Line style should be dashed
- Linewidth should be 2.

```python
import matplotlib.pyplot as plt

# Data
weight = [19.7, 21.3, 23.5, 25.9, 28.5, 32.1, 35.7, 39.6, 43.2]
height = [121.9, 124.5, 129.5, 134.6, 139.7, 147.3, 152.4, 157.5,
162.6]

# Plot settings
plt.figure(figsize=(8, 6))  # Adjust figure size if necessary
plt.plot(weight, height, color='green', marker='*', markersize=10,
linestyle='--', linewidth=2, label='Average Weight vs Height')

# Adding labels and title
plt.title('Average weight with respect to average height')
plt.xlabel('Weight in kg')
plt.ylabel('Height in cm')

# Display the grid
plt.grid(True)

# Adding legend
plt.legend()

# Show plot
plt.show()
```

The output will be:



In this example:

- plt.figure(figsize=(8, 6)): This function creates a new figure (or window) for the plot with a specified size. The figsize parameter takes a tuple (width, height) in inches. In this example, (8, 6) specifies a figure size of 8 inches wide and 6 inches high. Adjust these values according to your preference to control the size of the plot.
- Styling Parameters:
  - color='green': Sets the color of the line to green.
  - marker='*': Sets the marker style to asterisk (*).
  - markersize=10: Sets the size of the markers to 10.
  - linestyle='--': Sets the line style to dashed.
  - linewidth=2: Sets the width of the line to 2.
- Labels and Title:
  - plt.title('Average weight with respect to average height'): Adds a title to the plot.
  - plt.xlabel('Weight in kg'): Labels the x-axis.
  - plt.ylabel('Height in cm'): Labels the y-axis.
- Grid and Legend:
  - plt.grid(True): Displays a grid on the plot.
  - plt.legend(): Adds a legend based on the label parameter in plt.plot().

Understand the integration of Matplotlib with NumPy

Data Generation: NumPy arrays can be used to generate data for plotting. NumPy arrays serve as input data for Matplotlib plots.

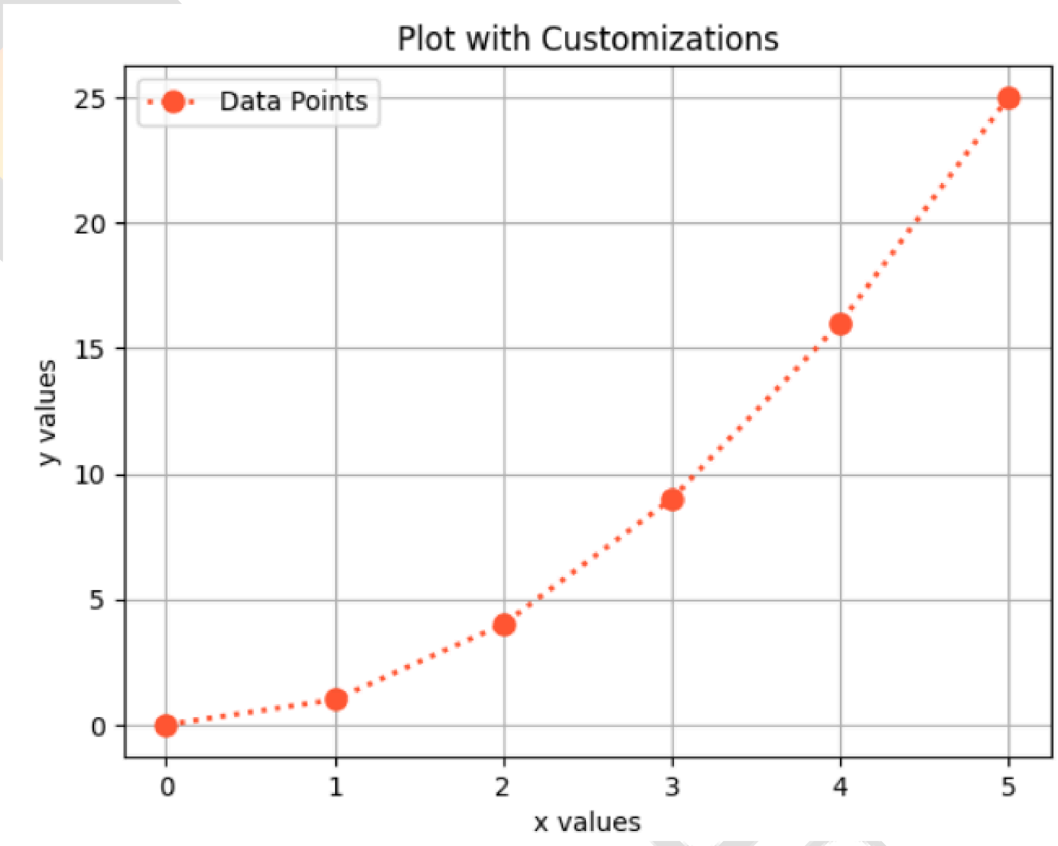
Efficient Plotting: Matplotlib directly accepts NumPy arrays as input for plotting. This integration ensures efficient handling and plotting of large datasets without the need for additional data format conversions.

Customization: NumPy arrays can be used to define custom tick positions, labels, and other parameters in Matplotlib plots. This customization enhances the appearance and clarity of visualizations.

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

# Generate data using NumPy arrays
x = np.array([0, 1, 2, 3, 4, 5])
y = x ** 2

# Create a plot using Matplotlib with specified marker, marker
size, line style, and line width
plt.plot(x, y, marker='o', markersize=8, linestyle=':',
linewidth=2, color='#FF5733', label='Data Points')

# Customize plot
plt.title('Plot with Customizations')
plt.xlabel('x values')
plt.ylabel('y values')
plt.grid(True)

# Customize x and y ticks
plt.xticks(np.arange(0, 6, step=1))  # Custom x ticks from 0 to 5
with step size 1
plt.yticks(np.arange(0, 26, step=5))  # Custom y ticks from 0 to
25 with step size 5

# Display legend
plt.legend()
```

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

The output will be:



In this example:
Generate Data:

x = np.array([0, 1, 2, 3, 4, 5]): Creates a NumPy array x containing integers from 0 to 5.

y = x ** 2: Computes the square of each element in x using NumPy's element-wise exponentiation to generate y.

Create Plot:

plt.plot(x, y, marker='o', markersize=8, linestyle=':', linewidth=2, color='#FF5733', label='Data Points'):
marker='o': Sets the marker style to a circle.
markersize=8: Sets the marker size to 8 points.
linestyle=':': Sets the line style to dotted.
linewidth=2: Sets the line width to 2 points.
color='#FF5733': Sets the color of the line and markers using a hexadecimal color code (#FF5733 corresponds to a shade of red-orange).

label='Data Points': Provides a label for the plot legend.

Understand the integration of Matplotlib with Pandas

We have three datasets given below:
Customers.csv
Purchase.csv
Products.csv

we need to import two libraries, Pandas and Matplotlib.

```
import pandas as pd
import matplotlib.pyplot as plt
```

```
Customers= pd.read_csv("Customers.csv")
Purchase=pd.read_csv("Purchase.csv")
Products = pd.read_csv("Products.csv")
```

Before visualizing data, it is essential to ensure that the DataFrame is clean, with no null values or other data quality issues. Clean data ensures that the visualizations are accurate and meaningful.

```
Customers.isnull().sum()
```

```
id               0
first_name       0
last_name        0
email            0
gender           0
street_num       0
street_name      0
street_suffix    0
city             0
state            0
postcode         0
dtype: int64
```

```
Purchase.isnull().sum()
```

```
id             0
purch_date     0
customer_num   0
product_num    0
amount         0
paid           0
dtype: int64
```

```
Products.isnull().sum()
```

```
id         0
product    0
cost       0
company    0
dtype: int64
```

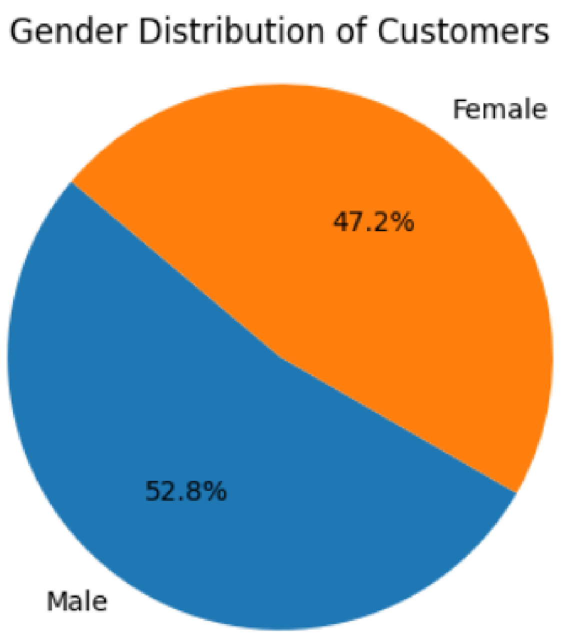
Since there are no missing values in your CSV files (Customers.csv, Purchase.csv, and Products.csv), we can proceed with visualizing the data using Matplotlib.

Visualizing data from the Customers.csv file can provide several insights depending on the types of plots and analyses we perform.

1. Gender Distribution:
   A pie chart or bar plot can show the distribution of customers by gender. This helps in understanding the gender demographics of customer base.

```
# Calculate gender distribution
gender_counts = Customers['gender'].value_counts()

# Plotting the pie chart
plt.figure(figsize= (4, 4))
plt.pie(gender_counts, labels=gender_counts.index,
autopct='%1.1f%%', startangle=140, colors=['#1f77b4', '#ff7f0e'])
plt.title('Gender Distribution of Customers')
plt.axis('equal')  # Ensure the pie is drawn as a circle
plt.show()
```

The output will be:



In this example:

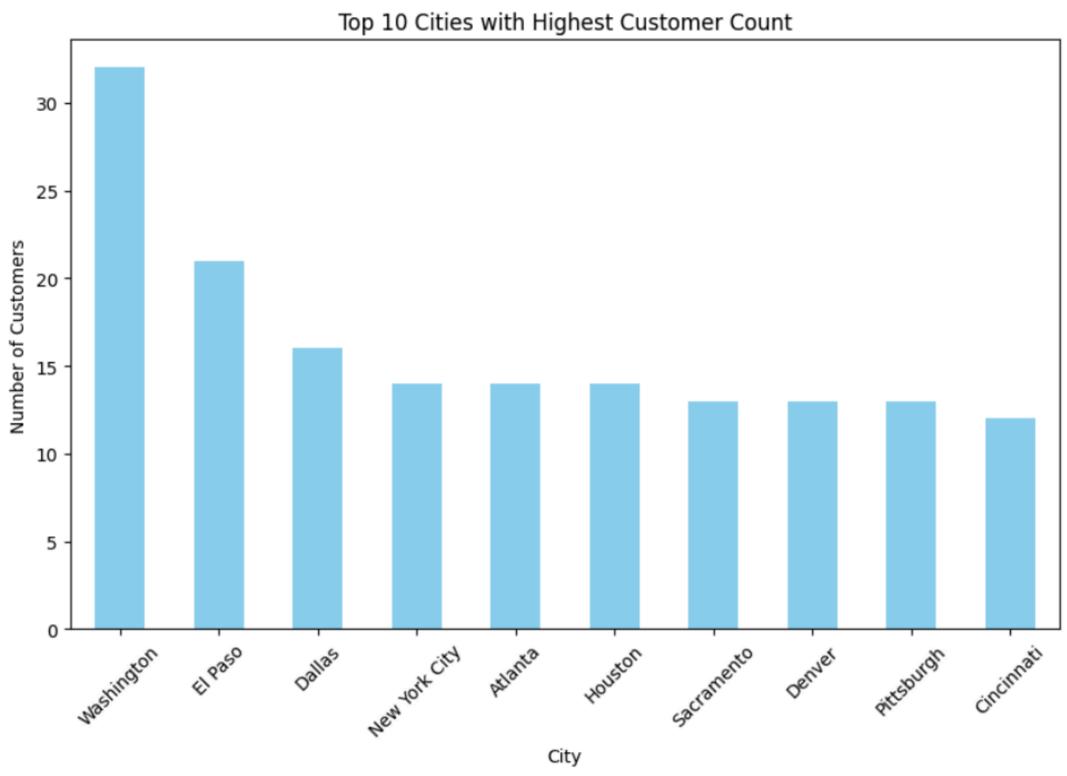
- Calculate Gender Distribution:
    - gender_counts = Customers['gender'].value_counts(): Count the occurrences of each gender category ('Male' and 'Female') in the DataFrame.
- Plotting the Pie Chart:
    - plt.figure(figsize=(4, 4)): Set the figure size.
    - plt.pie(...): Create a pie chart with:
    - gender_counts: Data values (counts of each gender).
    - labels=gender_counts.index: Labels for each slice ('Male' and 'Female').
    - autopct='%1.1f%%': Format for displaying the percentage on each slice.
    - startangle=140: Start angle for the first slice (helps in better visualization).
    - colors=['#1f77b4', '#ff7f0e']: Custom colors for 'Male' and 'Female'.
    - plt.title('Gender Distribution of Customers'): Set the title of the plot.
    - plt.axis('equal'): Ensure the pie chart is drawn as a circle for correct aspect ratio.
- Display the Plot:
    - plt.show(): Show the pie chart plot.

2. Location Analysis:
   Plots such as bar charts can show the distribution of customers by city, state, or region. This helps in understanding where your customers are located geographically.

```
# Calculate city-wise customer counts
city_counts = Customers['city'].value_counts().head(10)  # Top 10
cities by customer count
```

```
# Plotting the bar chart
plt.figure(figsize=(10, 6))
city_counts.plot(kind='bar', color='skyblue')
plt.title('Top 10 Cities with Highest Customer Count')
plt.xlabel('City')
plt.ylabel('Number of Customers')
plt.xticks(rotation=45)
plt.show()
```

The output will be:



In this example:

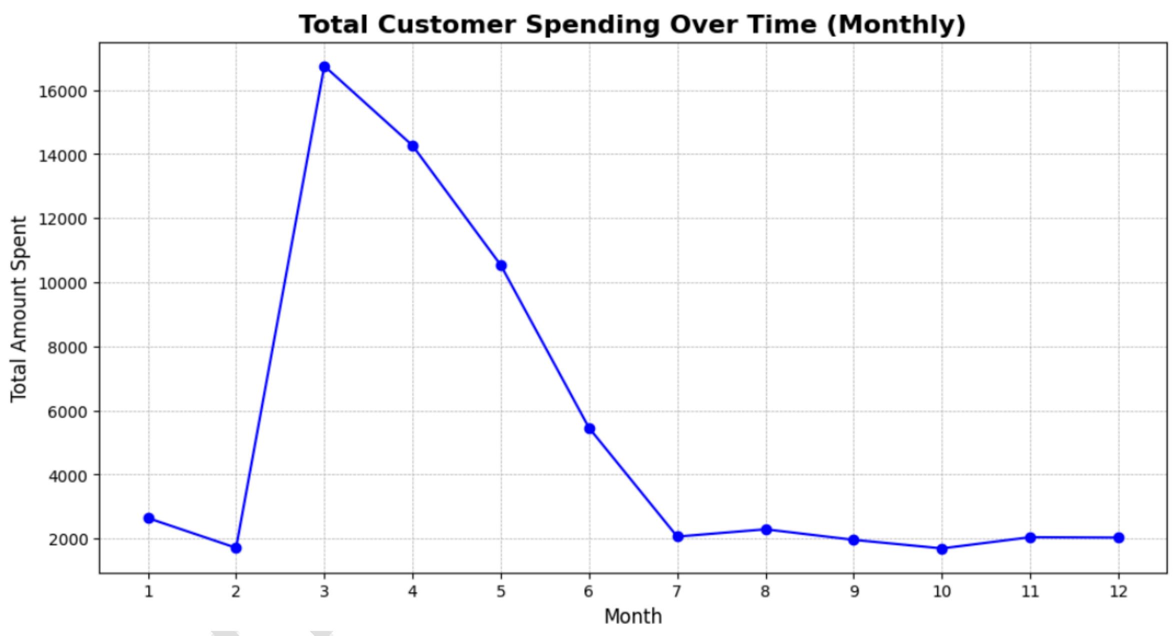
- Calculate City-wise Customer Counts:
  - city_counts = Customers['city'].value_counts().head(10): Count the occurrences of each city and select the top 10 cities by customer count. Adjust .head(10) based on the number of cities you want to visualize.
- Plotting the Bar Chart:
  - plt.figure(figsize=(10, 6)): Set the figure size.
  - city_counts.plot(kind='bar', color='skyblue'): Create a bar chart with:
  - kind='bar': Specify the type of plot as a bar chart.
  - color='skyblue': Set the color of the bars.
  - plt.title('Top 10 Cities with Highest Customer Count'): Set the title of the plot.
  - plt.xlabel('City'): Set the label for the x-axis.

- plt.ylabel('Number of Customers'): Set the label for the y-axis.
- plt.xticks(rotation=45): Rotate x-axis labels for better readability.
- plt.show(): Display the bar chart plot.

3. To visualize customer spending patterns over time on a monthly basis using a line plot or bar chart, we will need to use the Purchase.csv file.

```python
# Ensure the 'purch_date' column is in datetime format
Purchase['purch_date'] = pd.to_datetime(Purchase['purch_date'])

# Extract the month from the 'purch_date' column
Purchase['month'] = Purchase['purch_date'].dt.month

# Group by month and sum the amount
spending_over_time_monthly =
Purchase.groupby('month')['amount'].sum().reset_index()

# Plotting the line chart
plt.figure(figsize=(12, 6))
plt.plot(spending_over_time_monthly['month'],
spending_over_time_monthly['amount'], marker='o', linestyle='-',
color='b')

# Customize plot
plt.title('Total Customer Spending Over Time (Monthly)',
fontsize=16, fontweight='bold')
plt.xlabel('Month', fontsize=12)
plt.ylabel('Total Amount Spent', fontsize=12)
plt.grid(True, linestyle='--', linewidth=0.5)
plt.xticks(spending_over_time_monthly['month'])

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

The output will be:



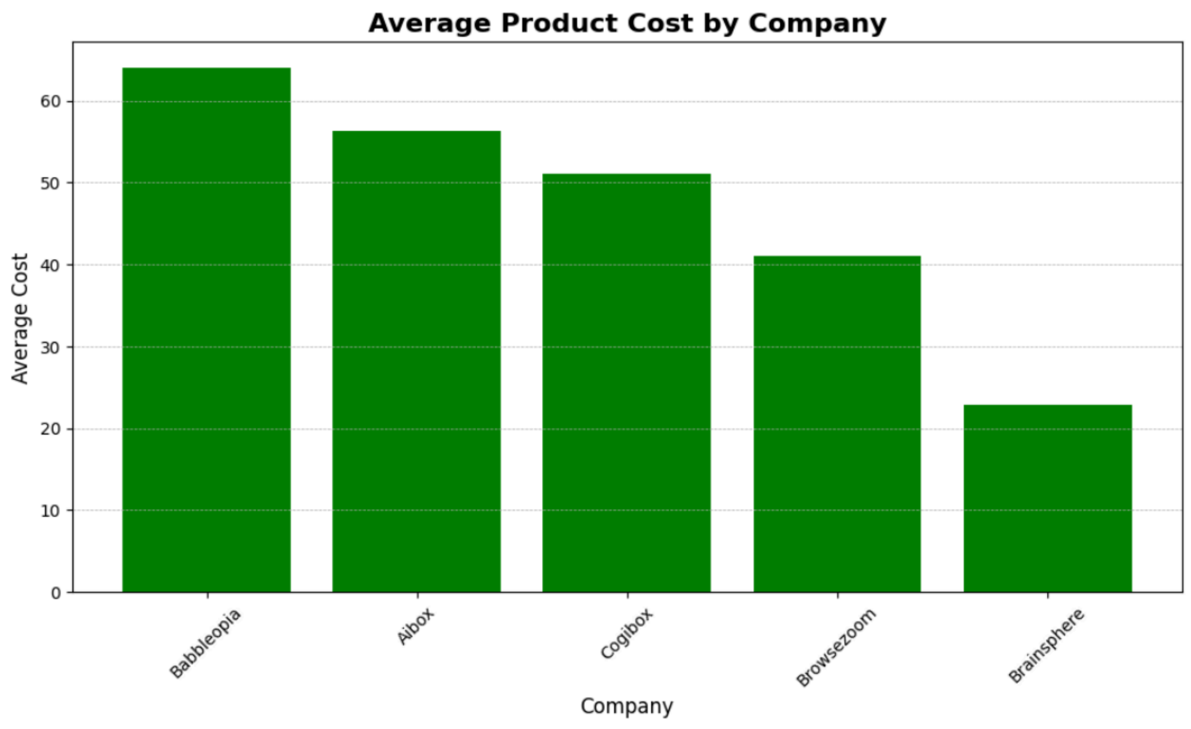
In this example:

- Convert 'purch_date' to Datetime:
  - Purchase['purch_date'] = pd.to_datetime(Purchase['purch_date']): Ensure the purch_date column is in datetime format for accurate time series plotting.
- Extract Month from 'purch_date':
  - Purchase['month'] = Purchase['purch_date'].dt.month: Extract the month from the purch_date column and store it in a new column month.
- Group by Month and Sum Amounts:
  - spending_over_time_monthly = Purchase.groupby('month')['amount'].sum().reset_index(): Group the data by month and sum the amount for each month.
- Plotting the Line Chart:
  - plt.figure(figsize=(20, 6)): Set the figure size.
  - plt.plot(...): Create a line plot with:
  - spending_over_time_monthly['month']: Months on the x-axis.
  - spending_over_time_monthly['amount']: Total amount spent on the y-axis.
  - marker='o': Mark data points with circles.
  - linestyle='-': Use solid lines between data points.
  - color='b': Set line color to blue.
  - plt.title(...): Set the title of the plot.
  - plt.xlabel(...): Set the label for the x-axis.
  - plt.ylabel(...): Set the label for the y-axis.
  - plt.grid(True, ...): Add a grid to the plot for better readability.
  - plt.xticks(spending_over_time_monthly['month']): Set custom x-axis ticks to match the months.
  - plt.show(): Show the line plot.

4. Cost Distribution by Company:
   A bar chart can show the average product cost for each company. This helps in understanding the pricing strategies of different companies and comparing the average costs across companies.

```
# Group by company and calculate the average cost
avg_cost_by_company =
Products.groupby('company')['cost'].mean().reset_index().head()

# Sort the data by cost for better visualization
avg_cost_by_company = avg_cost_by_company.sort_values(by='cost',
ascending=False)

# Plotting the bar chart
plt.figure(figsize=(12, 6))
plt.bar(avg_cost_by_company['company'],
avg_cost_by_company['cost'], color='Green')

# Customize plot
plt.title('Average Product Cost by Company', fontsize=16,
fontweight='bold')
plt.xlabel('Company', fontsize=12)
plt.ylabel('Average Cost', fontsize=12)
plt.xticks(rotation=45)
plt.grid(axis='y', linestyle='--', linewidth=0.5)

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

The output will be:



In this example:

- Group by Company and Calculate Average Cost:
    - avg_cost_by_company = products_df.groupby('company')['cost'].mean().reset_index().head(): Group the data by company and calculate the mean cost for each company. Reset the index for easier plotting.
- Sort Data:
    - avg_cost_by_company = avg_cost_by_company.sort_values(by='cost', ascending=False): Sort the DataFrame by the cost column in descending order to improve visualization clarity.
- Plotting the Bar Chart:
    - plt.figure(figsize=(12, 6)): Set the figure size.
    - plt.bar(...): Create a bar chart with:
    - avg_cost_by_company['company']: Companies on the x-axis.
    - avg_cost_by_company['cost']: Average costs on the y-axis.
    - color='skyblue': Set the bar color to sky blue.
    - plt.title(...): Set the title of the plot.
    - plt.xlabel(...): Set the label for the x-axis.
    - plt.ylabel(...): Set the label for the y-axis.
    - plt.xticks(rotation=45): Rotate x-axis labels for better readability.
    - plt.grid(axis='y', ...): Add a grid to the y-axis for better readability.
    - plt.show(): Show the bar chart.

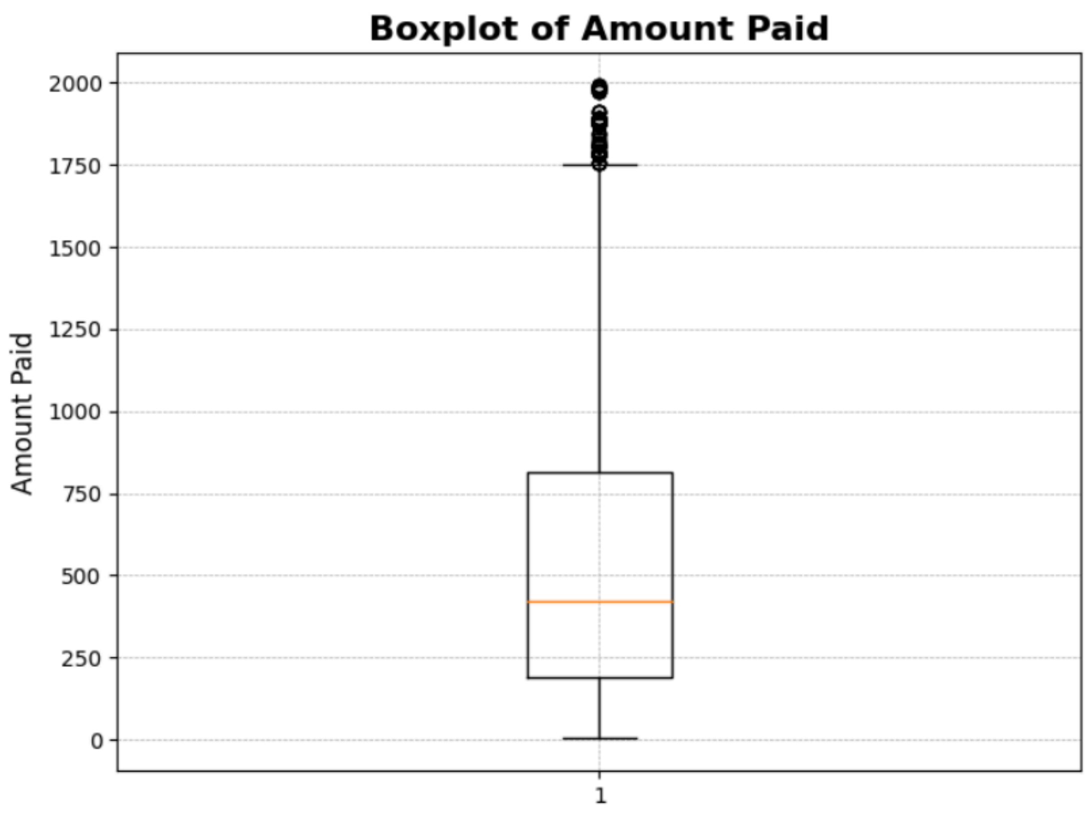
To plot a boxplot of the Paid column from the Purchase DataFrame, we can use the boxplot function from the matplotlib.pyplot module.

```python
# Create a boxplot of the 'Paid' column
plt.figure(figsize=(8, 6))
plt.boxplot(Purchase['paid'])

# Customize plot
plt.title('Boxplot of Amount Paid', fontsize=16,
fontweight='bold')
plt.ylabel('Amount Paid', fontsize=12)
plt.grid(True, linestyle='--', linewidth=0.5)

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

In this example:

- plt.figure(figsize=(8, 6)): Creates a new figure with a specified size (8 inches wide by 6 inches tall).
- plt.boxplot(Purchase['paid']): Creates a boxplot of the Paid column from the Purchase DataFrame.
- plt.title('Boxplot of Amount Paid', fontsize=16, fontweight='bold'): Sets the title of the plot with a font size of 16 and bold font weight.
- plt.ylabel('Amount Paid', fontsize=12): Labels the y-axis as "Amount Paid" with a font size of 12.
- plt.grid(True, linestyle='--', linewidth=0.5): Adds a grid to the plot with dashed lines and a line width of 0.5 for better readability.
- plt.show(): Displays the plot.

The output will be:



Interpretation of the Boxplot:

1. Median (Orange Line):
   The orange line in the box represents the median amount paid. This is the middle value of the data, where half of the values are above and half are below.
2. Whiskers:
   The "whiskers" extend from the quartiles to the smallest and largest values within 1.5 * IQR from the quartiles.Values beyond the whiskers are considered outliers.
3. Outliers (Circles):
   The circles above the upper whisker represent outliers. These are values that are significantly higher than the rest of the data.

We now turn our attention to Seaborn, a powerful data visualization library built on top of Matplotlib that provides a high-level interface for drawing attractive and informative statistical graphics.

Additional features in matplotlib (only if time permits. FYI for students)

- To save the output in a particular format and location
  - plt.savefig('plot.png', format='png')
- Regression Plot: Adding a regression line to a scatter plot.

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

```
# Load the tips dataset
tips = sns.load_dataset('tips')
sns.lmplot(x="total_bill", y="tip", data=tips)
plt.show()
```

- Pairplot: To visualize pairwise relationship in the dataset

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

# Load the iris dataset
iris = sns.load_dataset('tips')
sns.pairplot(tips, hue='time')
plt.show()
```

Understand the importance of Seaborn in data visualization

Seaborn is a crucial tool for data visualization in Python due to its ability to simplify complex plotting tasks and produce aesthetically pleasing and informative graphics. Built on top of Matplotlib, Seaborn provides a high-level interface that makes it easier to generate sophisticated visualizations with minimal code. Key advantages of using Seaborn include:

- Seaborn offers built-in themes and color palettes that enhance the visual appeal of plots, making them more attractive and easier to interpret.
- With Seaborn, complex visualizations such as heatmaps, violin plots, and pair plots can be created with simple, concise commands.
- Seaborn includes specialized functions for visualizing statistical relationships, distributions, and trends, which are essential for data analysis.
- Seaborn works seamlessly with Pandas data structures, allowing for direct plotting from DataFrames, which simplifies the workflow for data analysts and scientists.
- From basic line plots to advanced multi-plot grids, Seaborn covers a wide range of visualization needs, making it a versatile choice for any data visualization task.

Installing Seaborn

To install Seaborn, you simply need to type:

```
conda install seaborn
```

or

```
pip install seaborn
```

Importing Seaborn

For importing seaborn, you need to use the following command:

```
import seaborn as sns
```

Seaborn comes with several built-in datasets that you can use to practice and explore data visualization techniques. These datasets are primarily for demonstration and educational purposes. Some of the commonly used built-in datasets include "tips," "iris," "flights," and more.

Using the tips Dataset: The tips dataset contains information about tips received by waitstaff in a restaurant, including the total bill, the tip amount, the day of the week, the time of day, the size of the party, and more.

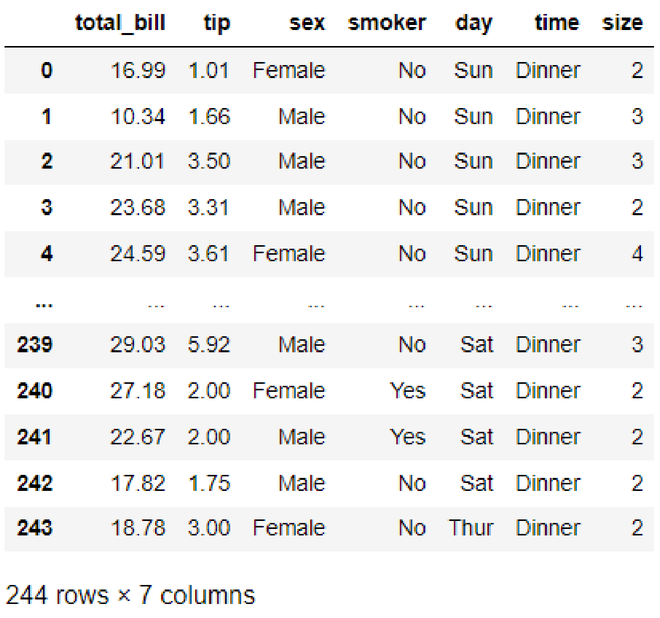
Understand common plot types in Seaborn

Importing necessary libraries

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

```
# Load the tips dataset
tips = sns.load_dataset("tips")
tips
```

The output will be:



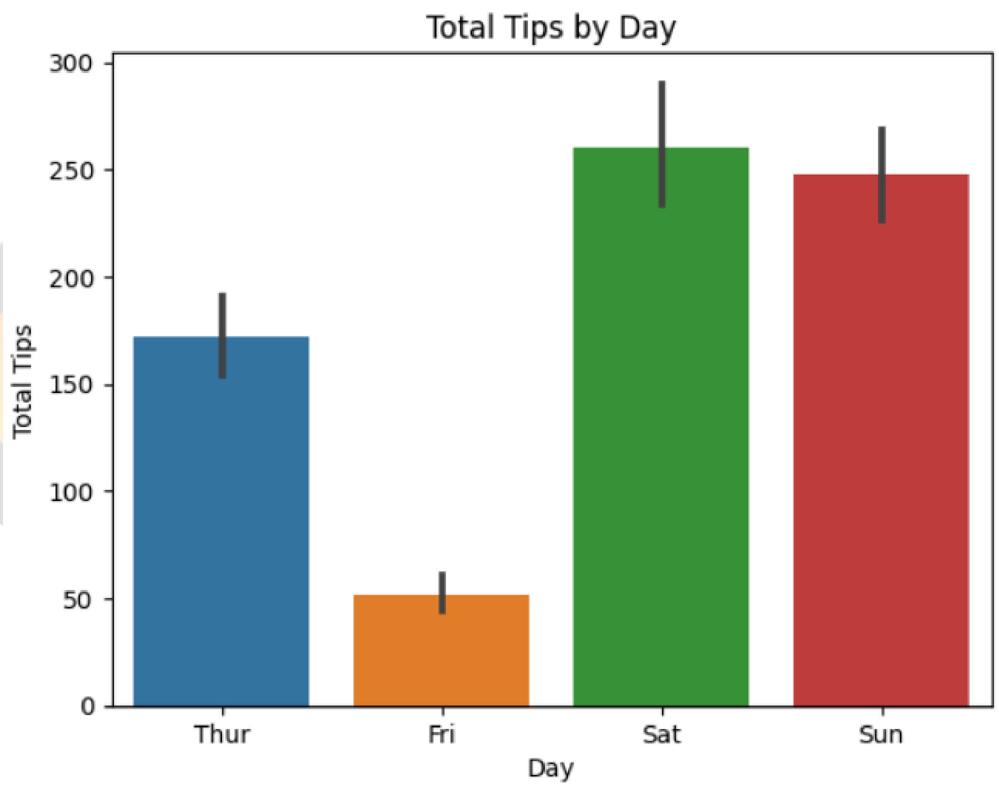
	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4
...	...	...	...	...	...	...	...
239	29.03	5.92	Male	No	Sat	Dinner	3
240	27.18	2.00	Female	Yes	Sat	Dinner	2
241	22.67	2.00	Male	Yes	Sat	Dinner	2
242	17.82	1.75	Male	No	Sat	Dinner	2
243	18.78	3.00	Female	No	Thur	Dinner	2

244 rows × 7 columns

1. Bar Plot: Bar plots can be useful for comparing quantities across different categories.

```
# Create a bar plot of total tips by day
sns.barplot(x="day", y="tip", data=tips, estimator=sum)
plt.title('Total Tips by Day')
plt.xlabel('Day')
plt.ylabel('Total Tips')
plt.show()
```

The output will be:



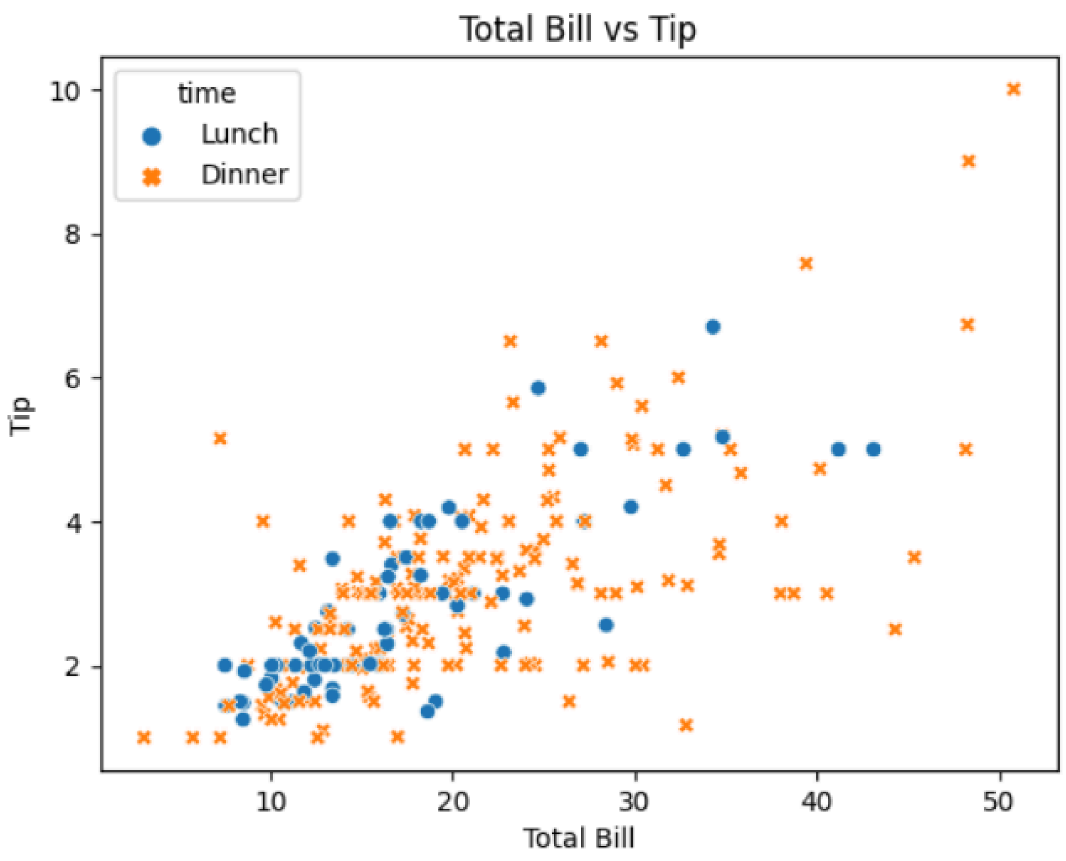
In this example:

- sns.barplot(): Creates a bar plot.
- x="day": The x-axis represents the days of the week.
- y="tip": The y-axis represents the tip amounts.
- data=tips: The dataset used is the tips dataset.
- estimator=sum: The function to apply to the y values, in this case, summing the tips for each day.
- plt.title(): Adds a title to the plot.
- plt.xlabel() and plt.ylabel(): Labels the x-axis and y-axis.

2. Scatter Plot: Scatter plots can help visualize the relationship between two numerical variables.

```
# Create a scatter plot of total bill vs tip
sns.scatterplot(x="total_bill", y="tip", hue="time", style="time",
data=tips)
plt.title('Total Bill vs Tip')
plt.xlabel('Total Bill')
plt.ylabel('Tip')
plt.show()
```

The output will be:



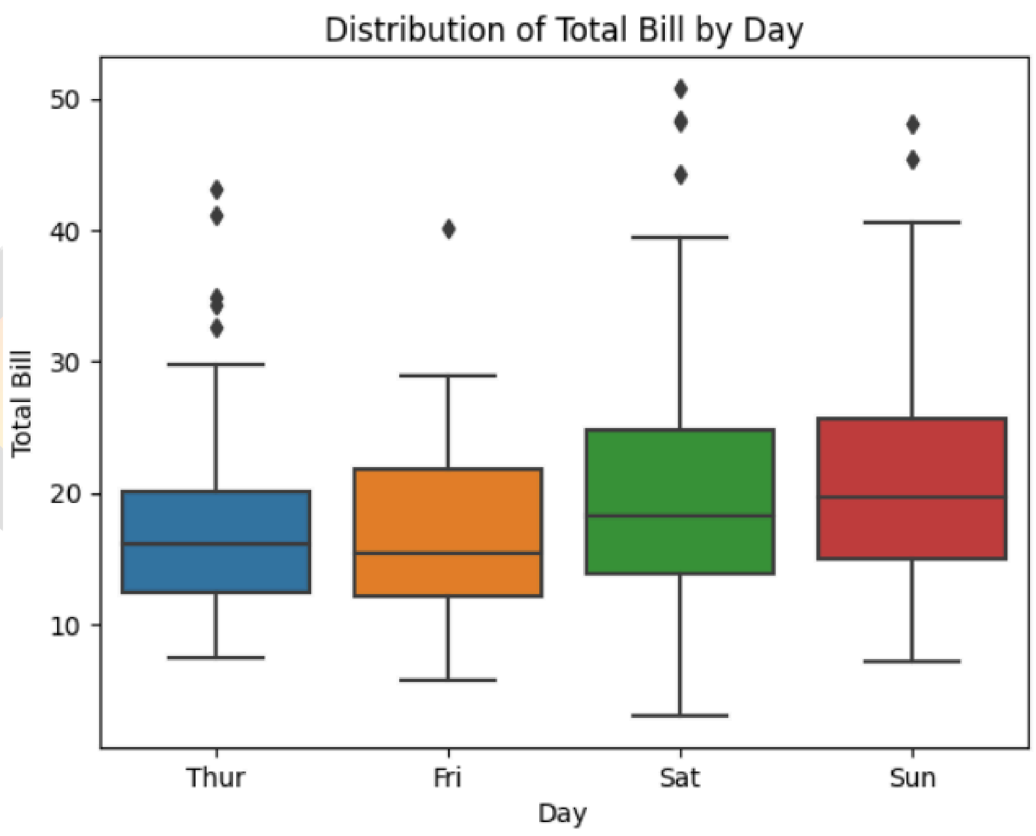
In this example:

- sns.scatterplot(): Creates a scatter plot.
- x="total_bill": The x-axis represents the total bill amount.
- y="tip": The y-axis represents the tip amount.
- hue="time": Colors the points based on the time of day (Lunch or Dinner).
- style="time": Changes the style of the points based on the time of day (Lunch or Dinner).
- data=tips: The dataset used is the tips dataset.
- plt.title(): Adds a title to the plot.
- plt.xlabel() and plt.ylabel(): Labels the x-axis and y-axis.

3. Box Plot: Box plots show the distribution of a variable and can highlight outliers.

```
# Create a box plot of total bill by day
sns.boxplot(x="day", y="total_bill", data=tips)
plt.title('Distribution of Total Bill by Day')
plt.xlabel('Day')
plt.ylabel('Total Bill')
plt.show()
```

The output will be:



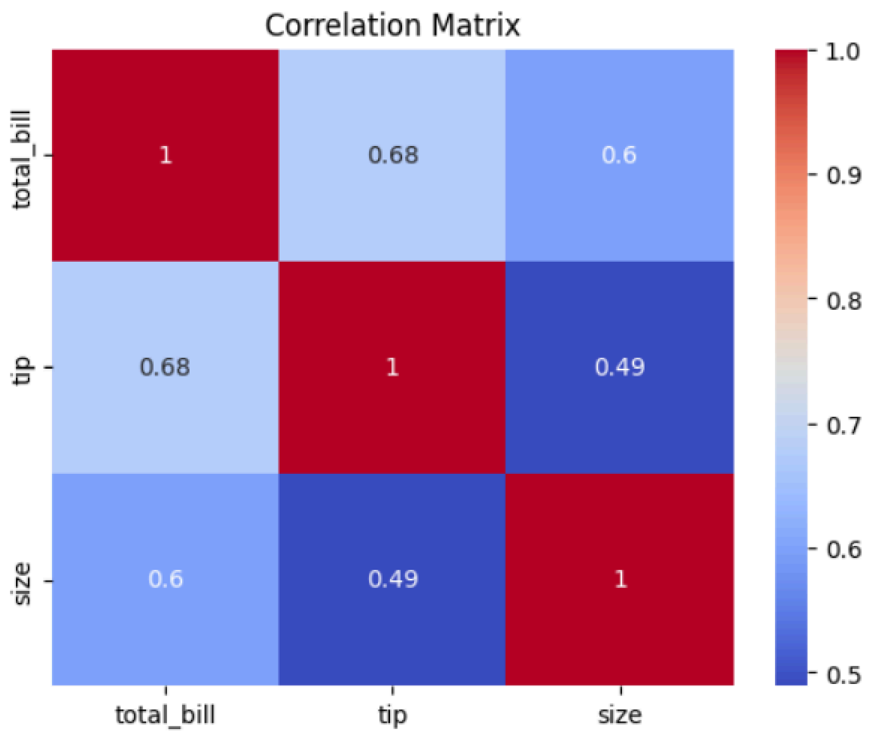
In this example:

- sns.boxplot(): Creates a box plot.
- x="day": The x-axis represents the days of the week.
- y="total_bill": The y-axis represents the total bill amounts.
- data=tips: The dataset used is the tips dataset.
- plt.title(): Adds a title to the plot.
- plt.xlabel() and plt.ylabel(): Labels the x-axis and y-axis.

4. Heatmap: Heatmaps can show the relationship between different variables.

```
# Select only numeric columns for correlation
numeric_tips = tips.select_dtypes(include='number')

# Compute the correlation matrix
corr = numeric_tips.corr()

# Create a heatmap of the correlation matrix
sns.heatmap(corr, annot=True, cmap="coolwarm")
plt.title('Correlation Matrix')
plt.show()
```

The output will be:



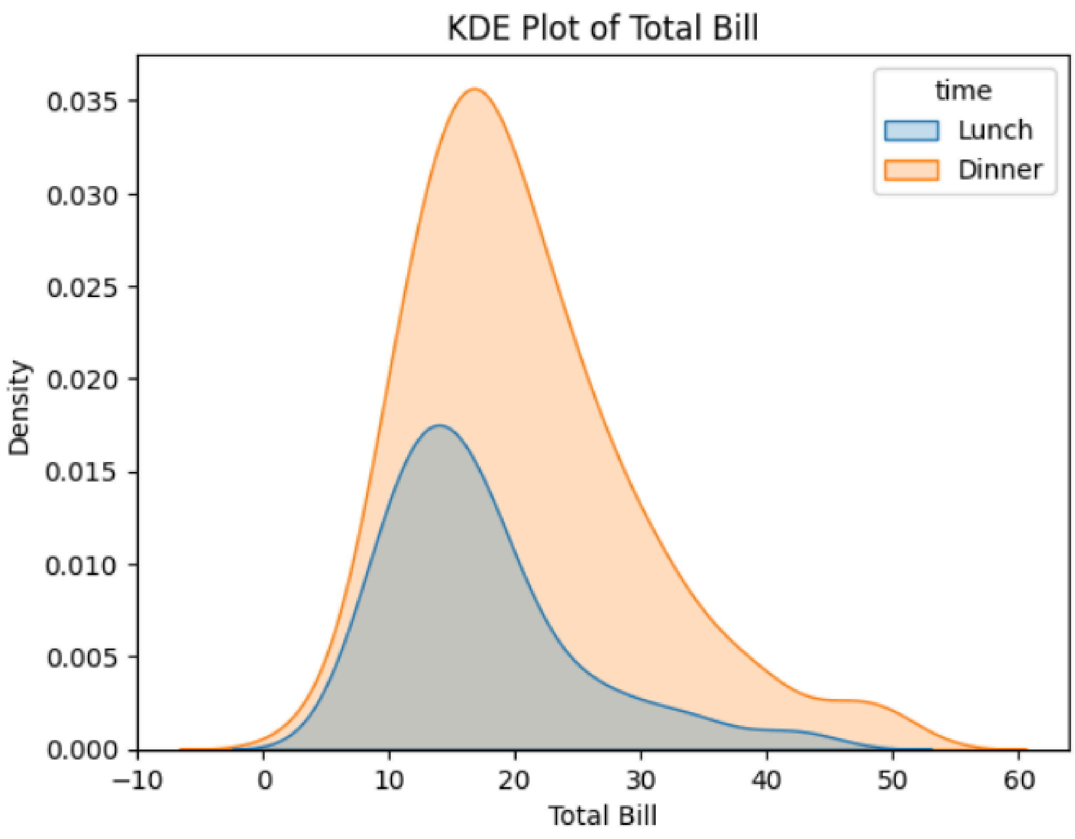
In this example:

- tips.select_dtypes(include='number'): Selects only the numeric columns from the tips dataset.
- numeric_tips.corr(): Computes the correlation matrix for these numeric columns.
- sns.heatmap(): Creates a heatmap.
    - corr: The correlation matrix.
    - annot=True: Annotates each cell in the heatmap with the correlation coefficient.
    - cmap="coolwarm": Specifies the color map to use for the heatmap.
- plt.title(): Adds a title to the plot.

5. KDE Plot: KDE plots are useful for visualizing the probability density of a variable.

```
# Create KDE plot of total bill with different colors for 'Lunch'
and 'Dinner'
sns.kdeplot(data=tips, x="total_bill", hue="time", fill=True)
plt.title('KDE Plot of Total Bill')
plt.xlabel('Total Bill')
plt.show()
```

The output will be:



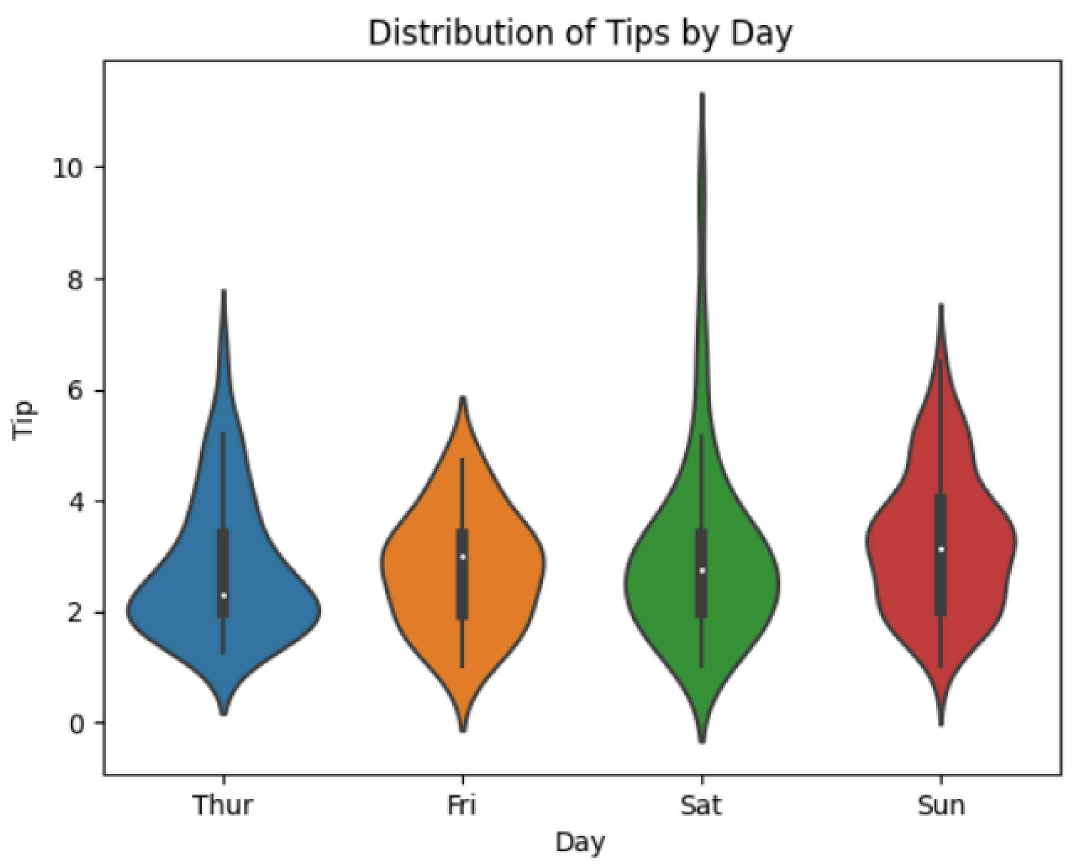
In this example:

- sns.kdeplot(): Creates a KDE plot.
- data=tips: Specifies the dataset (tips dataset in this case).
- x="total_bill": Specifies the variable for the x-axis (total bill).
- hue="time": Colors the KDE plot based on the 'time' variable ('Lunch' or 'Dinner').
- fill=True: Fills the area under the KDE curve with colors based on the hue variable.
- plt.title(): Adds a title to the plot.
- plt.xlabel(): Labels the x-axis.
- plt.show(): Displays the plot.

6. Violin Plot: Violin plots combine the features of box plots and KDE plots to show distributions.

```
# Create a violin plot of tips by day
sns.violinplot(x="day", y="tip", data=tips)
plt.title('Distribution of Tips by Day')
plt.xlabel('Day')
plt.ylabel('Tip')
plt.show()
```

The output will be:



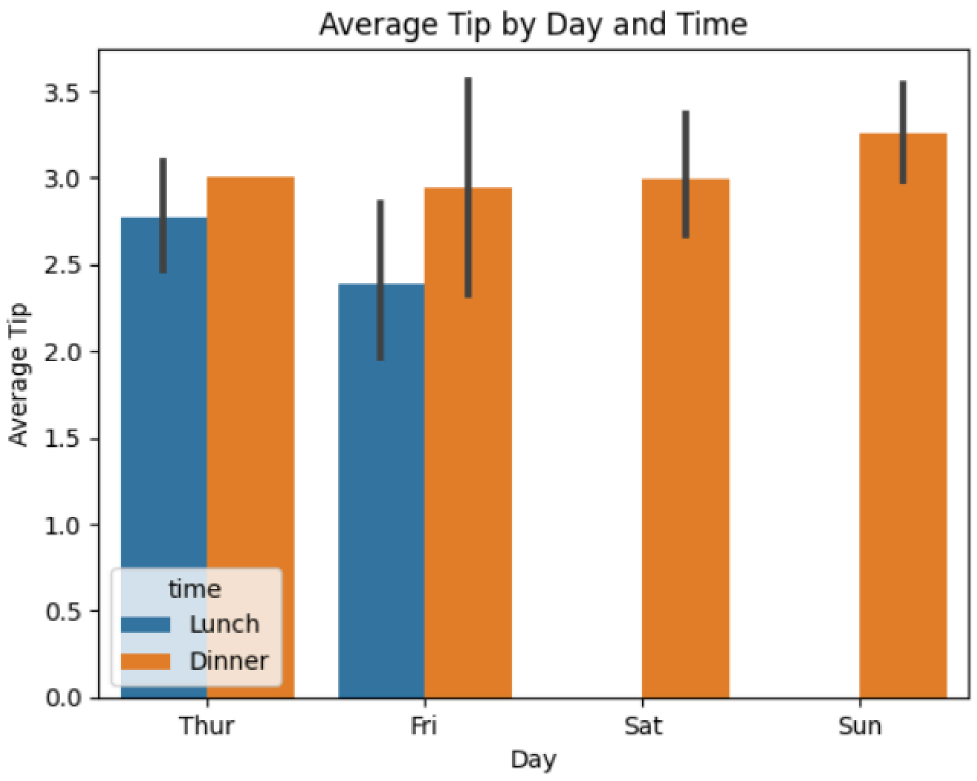
In this example:

- sns.violinplot(): Creates a violin plot.
- x="day": The x-axis represents the days of the week.
- y="tip": The y-axis represents the tip amounts.
- data=tips: The dataset used is the tips dataset.
- plt.title(): Adds a title to the plot.
- plt.xlabel() and plt.ylabel(): Labels the x-axis and y-axis.

7. Bar Plot with Multiple Categories

```
# Create a bar plot of average tip by day and time
sns.barplot(x="day", y="tip", hue="time", data=tips,
estimator="mean")
plt.title('Average Tip by Day and Time')
plt.xlabel('Day')
plt.ylabel('Average Tip')
plt.show()
```

The output will be:



Appendix

Name/Type of Plot	When is it Generally Used	Syntax to Create the Plot	Example with tips Dataset
Line Plot	To display trends over time or continuous data	plt.plot(x, y)	plt.plot(tips['total_bill'], tips['tip'])
Scatter Plot	To show the relationship between two variables	plt.scatter(x, y)	plt.scatter(tips['total_bill'], tips['tip'])
Bar Plot	To compare quantities across different categories	plt.bar(x, height)	plt.bar(tips['day'], tips['tip'].mean())
Horizontal Bar Plot	To compare quantities across categories (horizontally)	plt.barh(y, width)	plt.barh(tips['day'], tips['tip'].mean())
Histogram	To show the distribution of a dataset	plt.hist(x, bins)	plt.hist(tips['total_bill'], bins=10)
Box Plot	To display the distribution and identify outliers	plt.boxplot(data)	plt.boxplot(tips['total_bill'])

Pie Chart	To show the proportions of a whole	plt.pie(x, labels)	plt.pie(tips['day'].value_counts(), labels=tips['day'].unique())
Area Plot	To display cumulative totals over time or another continuous variable	plt.fill_between(x, y)	plt.fill_between(tips['size'], tips['total_bill'])
Hexbin Plot	To visualize the relationship between two variables with hexagonal bins	plt.hexbin(x, y, gridsize)	plt.hexbin(tips['total_bill'], tips['tip'], gridsize=20)
Error Bar Plot	To show data with error bars	plt.errorbar(x, y, yerr)	plt.errorbar(tips['total_bill'], tips['tip'], yerr=1)
Stack Plot	To show multiple datasets stacked on top of each other	plt.stackplot(x, y1, y2, ...)	plt.stackplot(tips['day'], tips['total_bill'], tips['tip'])
Stem Plot	To show discrete data points	plt.stem(x, y)	plt.stem(tips['total_bill'], tips['tip'])
Step Plot	To show stepwise data	plt.step(x, y)	plt.step(tips['total_bill'], tips['tip'])
Violin Plot	To show the distribution of data across several levels	plt.violinplot(data)	plt.violinplot(tips['total_bill'])
Heatmap	To show a matrix as a heatmap	plt.imshow(data, cmap)	sns.heatmap(tips.corr(), annot=True)

<u>**Different kinds of plots that can be created using seaborn**</u>

Name	When is it Generally Used	Syntax to Create the Plot	Example with tips Dataset
Bar Plot	To compare quantities across different categories	sns.barplot(x, y, data)	sns.barplot(x='day', y='tip', data=tips)
Count Plot	To show the count of observations in each categorical bin	sns.countplot(x, data)	sns.countplot(x='day', data=tips)
Box Plot	To display the distribution and identify outliers	sns.boxplot(x, y, data)	sns.boxplot(x='day', y='total_bill', data=tips)

Violin Plot	To show the distribution of data across several levels	sns.violinplot(x, y, data)	sns.violinplot(x='day', y='total_bill', data=tips)
Strip Plot	To show all observations for each categorical bin	sns.stripplot(x, y, data)	sns.stripplot(x='day', y='total_bill', data=tips)
Swarm Plot	To show all observations for each categorical bin, adjusted to avoid overlap	sns.swarmplot(x, y, data)	sns.swarmplot(x='day', y='total_bill', data=tips)
Scatter Plot	To show the relationship between two numerical variables	sns.scatterplot(x, y, data)	sns.scatterplot(x='total_bill', y='tip', data=tips)
Line Plot	To display trends over time or continuous data	sns.lineplot(x, y, data)	sns.lineplot(x='size', y='total_bill', data=tips)
Histogram	To show the distribution of a single variable	sns.histplot(x, data)	sns.histplot(x='total_bill', data=tips, bins=10)
KDE Plot	To estimate the probability density function of a continuous variable	sns.kdeplot(x, data)	sns.kdeplot(x='total_bill', data=tips, fill=True)
Pair Plot	To show pairwise relationships in a dataset	sns.pairplot(data)	sns.pairplot(tips)
Heatmap	To show a matrix as a heatmap	sns.heatmap(data)	sns.heatmap(tips.corr(), annot=True, cmap='coolwarm')
Joint Plot	To show a bivariate plot with marginal univariate plots	sns.jointplot(x, y, data)	sns.jointplot(x='total_bill', y='tip', data=tips)
Lmplot	To show a scatter plot with a linear regression model fit	sns.lmplot(x, y, data)	sns.lmplot(x='total_bill', y='tip', data=tips)
Regplot	To show a scatter plot with a regression line	sns.regplot(x, y, data)	sns.regplot(x='total_bill', y='tip', data=tips)
Residplot	To show residuals of a linear regression	sns.residplot(x, y, data)	sns.residplot(x='total_bill', y='tip', data=tips)
Catplot	To show different types of categorical plots	sns.catplot(x, y, data, kind)	sns.catplot(x='day', y='total_bill', data=tips, kind='violin')
FacetGrid	To show multiple plots based on a variable	g = sns.FacetGrid(data, col, row)	g = sns.FacetGrid(tips, col='time', row='smoker'); g.map(sns.scatterplot, 'total_bill', 'tip')

PairGrid	To show pairwise relationships with customized plots	g = sns.PairGrid(data)	g = sns.PairGrid(tips); g.map(sns.scatterplot)