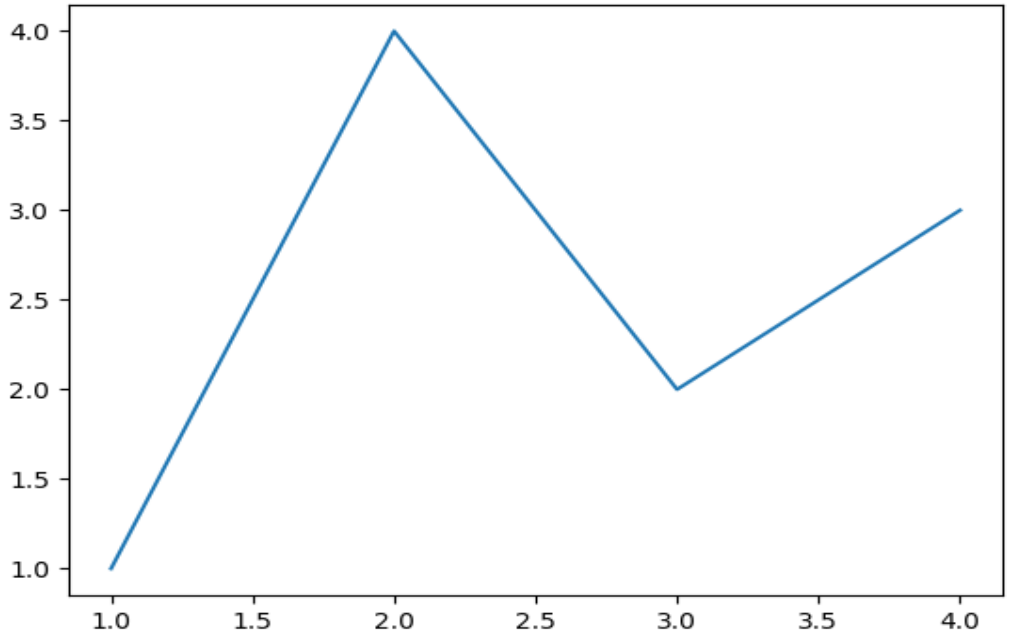
MATPLOTLIB

Matplotlib graphs your data on Figures (e.g., windows, Jupyter widgets, etc.), each of which can contain one or more Axes, an area where points can be specified in terms of x-y coordinates (or theta-r in a polar plot, x-y-z in a 3D plot, etc.). The simplest way of creating a Figure with an Axes is using pyplot.subplots. We can then use Axes.plot to draw some data on the Axes, and show() to display the figure.



Depending on the environment you are working in, plt.show() can be left out. This is for example the case with Jupyter notebooks, which automatically show all figures created in a code cell.

Types of inputs to plotting functions

Plotting functions expect numpy.array or numpy.ma.masked_array as input, or objects that can be passed to numpy.asarray. Classes that are similar to arrays ('array-like') such as pandas data objects and numpy.matrix may not work as intended. Common convention is to convert these to numpy.array objects prior to plotting. For example, to convert a numpy.matrix:

```
b = np.matrix([[1, 2], [3, 4]])
b_asarray = np.asarray(b)
```

Most methods will also parse a string-indexable object like a *dict*, a structured numpy array, or a pandas.DataFrame. Matplotlib allows you to provide the data keyword argument and generate plots passing the strings corresponding to the *x* and *y* variables.

Types of Plots in Matplotlib library:

1] Line Plot: Displays data as a series of points connected by straight lines. Useful for showing trends over time.

2] Scatter Plot: Uses dots to represent values for two different numeric variables. Useful for identifying relationships between variables.

3] Bar Chart: Represents categorical data with rectangular bars. Useful for comparing different groups or categories.

4] Histogram: Shows the distribution of a continuous variable by dividing the data into bins. Useful for visualizing the frequency distribution of a dataset.

5] Pie Chart: Circular chart divided into sectors, illustrating numerical proportions. Useful for showing parts of a whole.

6] Box Plot: Displays the distribution of data based on a five-number summary (minimum, first quartile, median, third quartile, and maximum). Useful for identifying outliers and the spread of the data.

7] Area Plot: Similar to line plots but the area below the line is filled. Useful for showing cumulative totals over time.

8] Heatmap: Displays data in matrix form using colors to represent values. Useful for visualizing data with two categorical axes.

Advantages:

Matplotlib is a powerful and versatile library for data visualization in Python, offering several key advantages:

1. **Comprehensive Plotting Capabilities:** Matplotlib supports a wide range of plot types, including line plots, scatter plots, bar charts, histograms, pie charts, and more, making it suitable for diverse visualization needs.
2. **Customizability:** Users can extensively customize plots with Matplotlib, adjusting elements like colors, labels, legends, and styles to create publication-quality graphics that meet specific requirements.
3. **Integration with Other Libraries:** Matplotlib integrates seamlessly with other Python libraries such as NumPy and pandas, facilitating easy data manipulation and visualization within a unified workflow.

Disadvantages:

Despite its strengths, Matplotlib has a few notable disadvantages:

1. **Complexity for Advanced Plots:** Creating complex and highly customized visualizations can be cumbersome and require a significant amount of code, making it less user-friendly for advanced plotting compared to some other libraries.
2. **Performance Issues with Large Datasets:** Matplotlib can become slow and less efficient when handling very large datasets or producing interactive visualizations, which can be a limitation for big data applications.
3. **Steeper Learning Curve:** Beginners might find Matplotlib's extensive options and parameters overwhelming, making it harder to learn compared to some higher-level plotting libraries like Seaborn or Plotly.

When and why to use the matplotlib library:

Matplotlib is best used in the following scenarios, due to its unique advantages and features:

1. Basic to Intermediate Visualizations:

- When: When you need to create standard plots such as line plots, scatter plots, bar charts, histograms, and pie charts.

- Why: Matplotlib provides straightforward functions for these plots, making it easy to generate basic visualizations quickly and effectively.

2. Publication-Quality Graphics:

- When: When you require highly customizable and publication-quality visualizations.

- Why: Matplotlib offers extensive customization options for plot elements, allowing precise control over the appearance of the final graphic, which is crucial for research papers and professional reports.

3. Integration with Other Python Libraries:

- When: When working within a Python ecosystem, especially with libraries like NumPy and pandas for data manipulation and analysis.

- Why: Matplotlib integrates seamlessly with these libraries, making it easy to plot data directly from pandas DataFrames or NumPy arrays.

4. Learning and Teaching Data Visualization:

- When: When teaching or learning the basics of data visualization and plotting in Python.

- Why: Matplotlib is often considered the foundation of Python plotting libraries, making it a good starting point to understand the principles of data visualization before moving on to more advanced tools.

Line Plot

A line plot is a type of chart that displays data points connected by straight lines, typically used to visualize trends over time or ordered categories. The x-axis represents the independent variable, often time or sequential categories, while the y-axis represents the dependent variable. Each point on the plot corresponds to a data value, with lines connecting the points to show the progression of data. Line plots are especially useful for identifying patterns, trends, and changes over intervals, making them a common choice for time-series data analysis. They can also compare multiple data series by plotting multiple lines on the same axes, allowing for easy visual comparison.

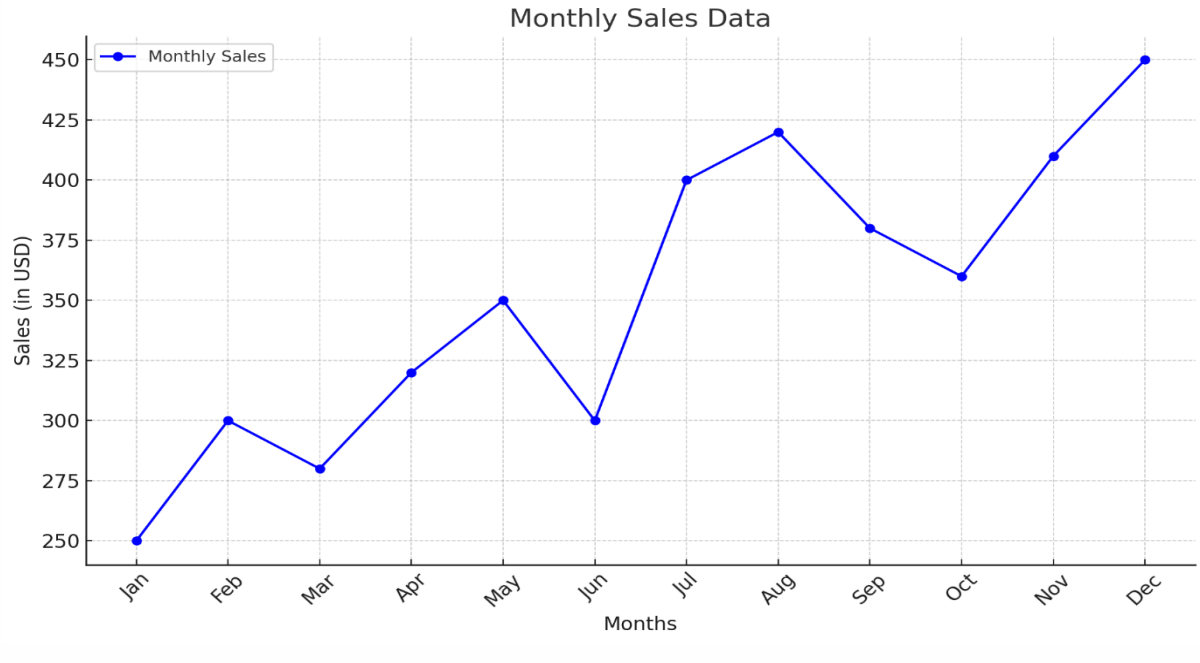
Here are several use cases for line plots:

1. **Sales Performance Over Time:**
    - **Use Case:** Visualize monthly sales data for a company to identify trends, seasonal patterns, and growth rates.
    - **Benefit:** Helps businesses track performance, forecast future sales, and make informed strategic decisions.
2. **Website Traffic Analysis:**
    - **Use Case:** Track daily website visits over a year to monitor user engagement and the impact of marketing campaigns.
    - **Benefit:** Enables digital marketers to understand visitor behavior, optimize content strategies, and improve user retention.
3. **Temperature Trends Monitoring:**
    - **Use Case:** Plot daily average temperatures over a year to study seasonal changes and long-term climate trends.
    - **Benefit:** Useful for meteorologists and researchers to analyze weather patterns and predict future climatic conditions.
4. **Health and Fitness Tracking:**
    - **Use Case:** Track a person's daily steps or workout duration over several months to monitor fitness progress.
    - **Benefit:** Assists individuals in maintaining their fitness goals and identifying periods of high or low activity.
5. **Stock Market Analysis:**
    - **Use Case:** Visualize the daily closing prices of a stock over multiple years to identify trends and investment opportunities.
    - **Benefit:** Provides investors and analysts with insights into stock performance, aiding in investment decisions.

Code Snippet:

```
import matplotlib.pyplot as plt

months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']

sales = [250, 300, 280, 320, 350, 300, 400, 420, 380, 360, 410, 450]

plt.figure(figsize=(10, 6))
```

```
plt.plot(months, sales, marker='o', linestyle='-', color='b', label='Monthly Sales')
plt.title('Monthly Sales Data')
plt.xlabel('Months')
plt.ylabel('Sales (in USD)')
plt.grid(True)
plt.legend()
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



Description:

- **Data:** The months list represents the months of the year, and the sales list represents the sales figures for each month.
- **Plotting:** plt.plot() creates the line plot with markers (marker='o'), solid lines (linestyle='-'), and a blue color (color='b').
- **Customization:** Titles and labels are added with plt.title(), plt.xlabel(), and plt.ylabel(). A grid is enabled with plt.grid(True), and a legend is added with plt.legend(). The x-axis labels are rotated for better readability using plt.xticks(rotation=45).
- **Display:** The plot is adjusted for tight layout and displayed using plt.tight_layout() and plt.show().

This code will generate a clear and readable line plot of monthly sales data.

Scatter Plot

A scatter plot in the matplotlib library is a type of plot that displays points representing individual data values. Each point's position on the horizontal (x-axis) and vertical (y-axis) dimensions reflects its values in the dataset, making it useful for observing relationships or correlations between variables. Matplotlib's scatter() function allows customization of point size, color, and style, enhancing the visual distinction between different data categories or groups. Scatter plots are particularly effective for highlighting trends, patterns, and potential outliers in bivariate data.

Scatter plots in the matplotlib library have multiple use cases across various fields:

1. Correlation Analysis: They are commonly used to visualize relationships between two variables, helping to identify trends or correlations. For example, examining the relationship between temperature and ice cream sales.

2. Outlier Detection: Scatter plots can reveal outliers as data points that significantly deviate from the general pattern of the data, aiding in anomaly detection.

3. Clustering Analysis: In machine learning and data mining, scatter plots can be used to visualize clusters of data points, such as in k-means clustering.

4. Distribution Patterns: They provide insights into the distribution of data points across the plot, indicating whether the data is concentrated or spread out.

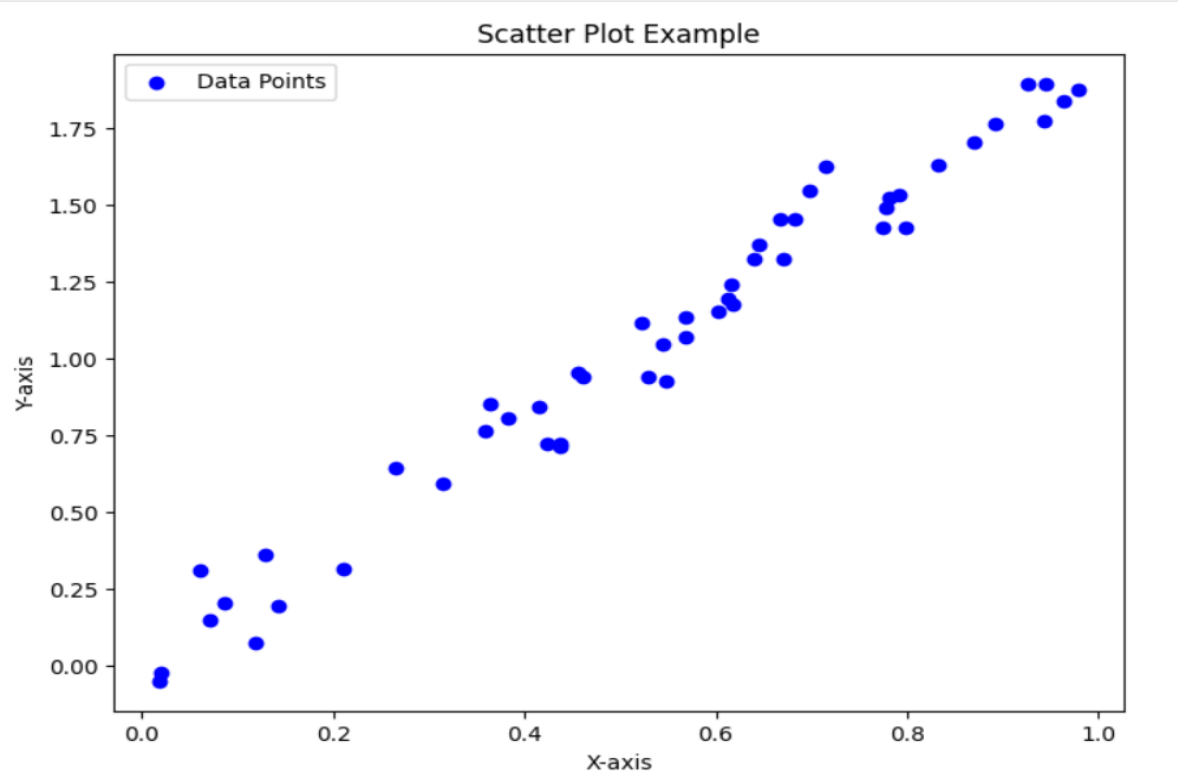
5. Comparison of Models: Scatter plots can be used to compare the outputs of different models or algorithms by plotting their predictions against actual observed values.

6. Experimental Data Analysis: Scientists often use scatter plots to visualize experimental results, helping to assess the effectiveness of interventions or treatments.

Code Snippet:

```
import matplotlib.pyplot as plt

import numpy as np

np.random.seed(0)

x = np.random.rand(50)

y = 2 * x + np.random.normal(0, 0.1, 50)

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

plt.scatter(x, y, color='blue', label='Data Points')

plt.title('Scatter Plot Example')

plt.xlabel('X-axis')

plt.ylabel('Y-axis')
```

```
plt.legend()
plt.show()
```



Description:

- **Import Libraries**: First, import matplotlib.pyplot as plt and numpy as np.
- **Generate Sample Data**: Using numpy.random.rand() to generate random values for x and y. y is created as a linear function of x plus some normally distributed noise.
- **Create Scatter Plot**: Use plt.scatter() to create the scatter plot. Customize the color of the markers (color='blue') and add a label for the legend (label='Data Points').
- **Customize Labels and Title**: Set the title of the plot (plt.title()), labels for the x-axis and y-axis (plt.xlabel() and plt.ylabel()), and add a legend (plt.legend()).
- **Display the Plot**: Finally, use plt.show() to display the scatter plot.

Bar Plot

A bar plot in data visualization is a graphical representation where rectangular bars of lengths proportional to the values they represent are used to display categorical data. Each bar typically corresponds to a category, and the height of the bar indicates the numerical value associated with that category. Bar plots are effective for comparing discrete categories against their respective values, making them useful for visualizing distributions, trends, and comparisons across different groups or time periods in datasets. They often include labels for both the x-axis (categories) and y-axis (values), providing clear insights into the relationships and differences between the categories being compared.

Bar plots in data visualization serve various practical purposes across different domains:

1. Comparison of Categories: They are widely used to compare the magnitude of categorical variables. For instance, comparing sales figures of different products or the performance of different teams in a competition.

2. Trend Analysis: Bar plots can illustrate trends over time or across different segments. For example, plotting monthly sales figures to identify seasonal trends or comparing demographic data across different years.

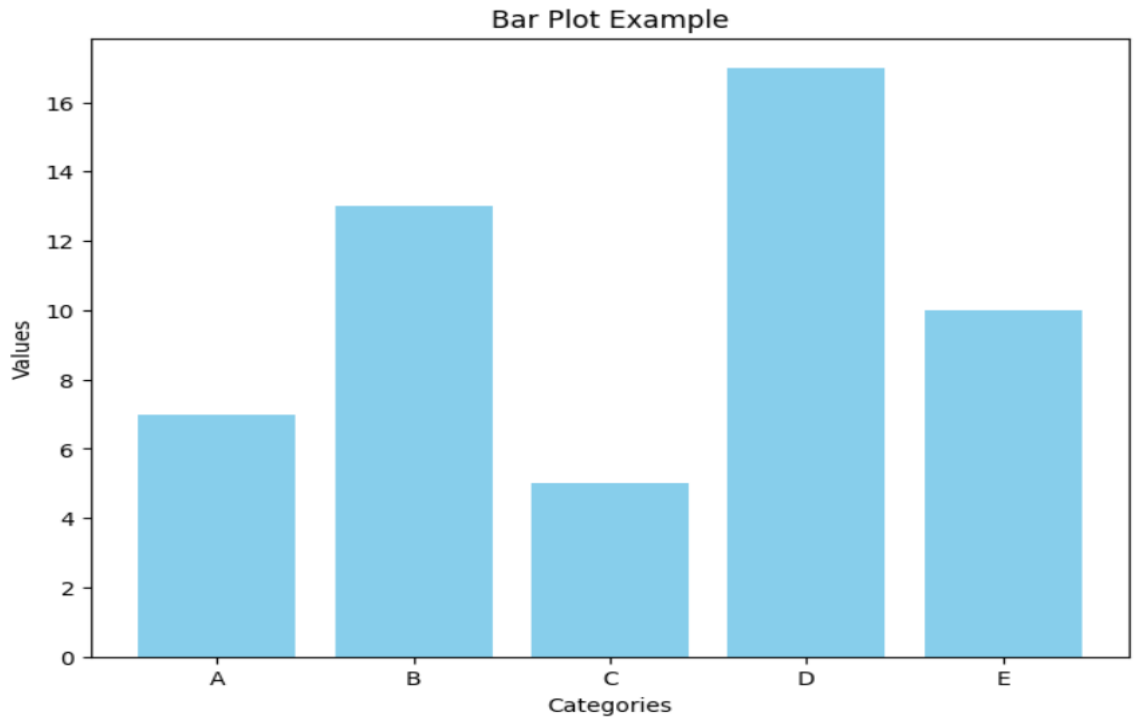
3. Distribution Visualization: They are effective in visualizing the distribution of data within categories, such as frequency distributions or histograms of discrete variables like age groups or income brackets.

4. Ranking and Sorting: Bar plots can display rankings or sorting of categories based on their values, providing a clear hierarchy or order of importance. This is useful in decision-making processes or prioritization tasks.

5. Comparison of Models or Scenarios: In analytics and modeling, bar plots can be used to compare the outcomes or predictions of different models or scenarios, helping to make informed decisions based on the observed differences.

Code Snippet:

```
import matplotlib.pyplot as plt

categories = ['A', 'B', 'C', 'D', 'E']

values = [7, 13, 5, 17, 10]

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

plt.bar(categories, values, color='skyblue')

plt.title('Bar Plot Example')

plt.xlabel('Categories')

plt.ylabel('Values')

plt.show()
```



Description:

1. **Imports:** import matplotlib.pyplot as plt: Imports the matplotlib plotting module.
2. **Data for the Bar Plot:** Categories: List of categories (labels) for the x-axis. Values: Corresponding numerical values for each category, which determine the height of the bars on the y-axis.
3. **Create Bar Plot:** plt.figure(figsize=(8, 6)): Creates a new figure with a specific size of 8 inches by 6 inches. plt.bar(categories, values, color='skyblue'): Plots a bar plot where each category from categories is associated with a bar whose height is determined by the values in values. The color='skyblue' argument sets the color of the bars.
4. **Customize Plot Labels and Title:** plt.title('Bar Plot Example'): Sets the title of the plot to "Bar Plot Example". plt.xlabel('Categories'): Labels the x-axis as "Categories". plt.ylabel('Values'): Labels the y-axis as "Values".
5. **Display the Plot:** plt.show(): Displays the complete bar plot with the specified title, labels, and colored bars.

Histogram

A histogram in Matplotlib is a graphical representation of the distribution of numerical data. It organizes a dataset into bins or intervals and counts the number of observations that fall into each bin, displaying this count with bars. The height of each bar represents the frequency or count of data points within each bin. Matplotlib's hist function is used to create histograms, allowing customization of bin sizes, colors, and other visual aspects. Histograms are useful for visualizing the underlying frequency distribution of a dataset, spotting skewness, and identifying potential outliers.

Histograms in Matplotlib have a wide range of use cases across various fields. Here are a few examples:

1. Data Analysis and Exploration: Histograms help in understanding the distribution of data points within a dataset, which is crucial for preliminary data analysis and identifying underlying patterns.

2. Quality Control: In manufacturing, histograms can be used to monitor the variation in product dimensions or other quality metrics, ensuring that they meet specified standards.

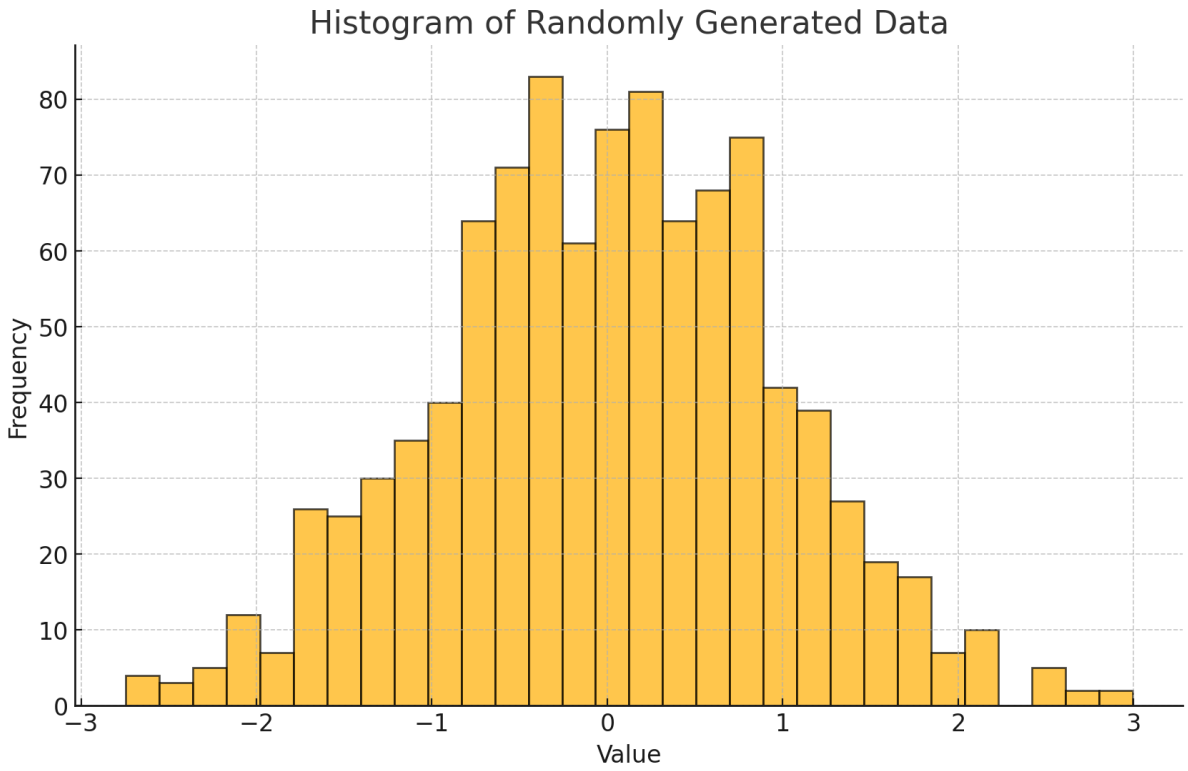
3. Finance: Analysts use histograms to visualize the distribution of financial metrics such as returns, prices, and trading volumes, aiding in risk assessment and decision-making.

4. Healthcare: In medical research, histograms can illustrate the distribution of patient vital signs or lab results, helping to detect abnormalities or trends.

5. Education: Teachers and educators use histograms to analyze the distribution of exam scores or other performance metrics, identifying areas where students might need additional support.

Code Snippet:

```
import matplotlib.pyplot as plt

import numpy as np

data = np.random.randn(1000)

plt.hist(data, bins=30, color='blue', edgecolor='black', alpha=0.7)

plt.title('Histogram of Normally Distributed Data')

plt.xlabel('Value')

plt.ylabel('Frequency')

plt.show()
```



Description:

- **Data Generation**: np.random.randn(1000) generates 1000 random data points from a standard normal distribution.
- **Creating the Histogram**: plt.hist(data, bins=30, edgecolor='black', alpha=0.7) creates the histogram with 30 bins. edgecolor='black' adds a black border to the bars, and alpha=0.7 makes the bars slightly transparent.
- **Adding Titles and Labels**: plt.title, plt.xlabel, and plt.ylabel are used to add a title and axis labels to the plot.
- **Displaying the Histogram**: plt.show() renders the histogram.

Pie Chart

A pie chart is a circular statistical graphic divided into slices to illustrate numerical proportions. Each slice of the pie represents a category's contribution to the whole, with the size of each slice proportional to the category's percentage of the total. Pie charts are useful for displaying relative sizes of parts to a whole, making it easy to see which categories dominate or contribute less. They are commonly used in business, media, and education to present survey results, financial data, and market shares. However, they are less effective for comparing the size of individual slices, especially when there are many categories or the differences are subtle.

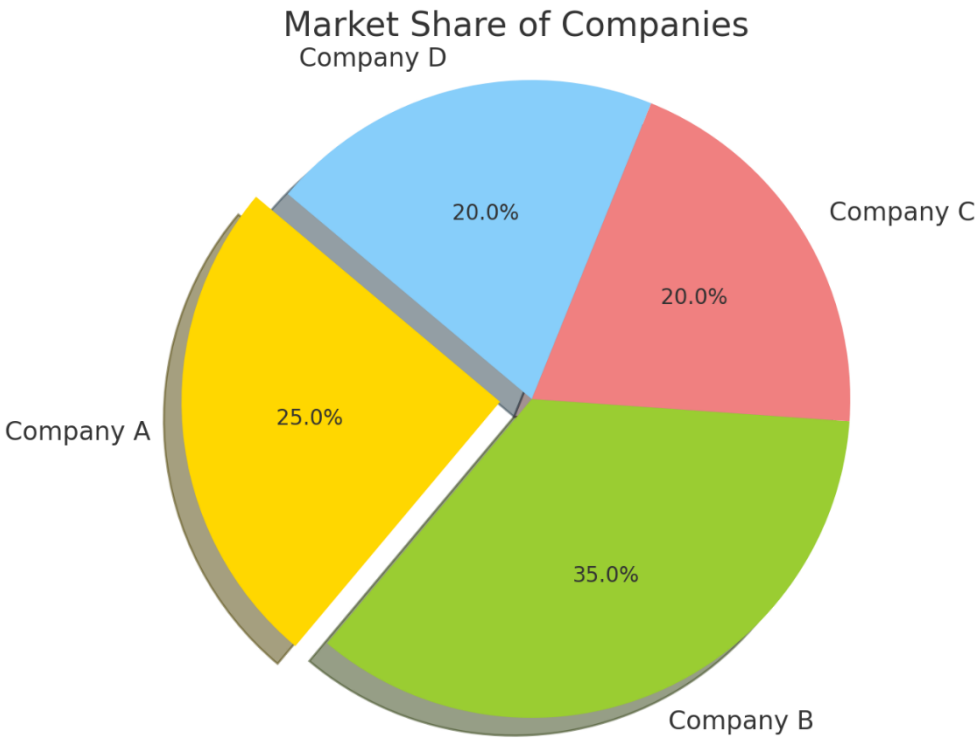
Pie charts have various use cases across different fields, serving as an effective tool for visualizing proportional data. Here are some examples:

1. **Market Share Analysis**: Businesses use pie charts to display the market share of different companies within an industry, helping stakeholders understand competitive dynamics.
2. **Budget Allocation**: Financial analysts and organizations use pie charts to illustrate how a budget is divided among different departments or expense categories.
3. **Survey Results**: Researchers and marketers use pie charts to represent the distribution of responses in survey data, making it easy to visualize the proportion of different answers.
4. **Sales Distribution**: Companies use pie charts to show the distribution of sales across various products, regions, or sales channels, aiding in strategic planning.
5. **Demographic Breakdown**: Demographers and sociologists use pie charts to present population distribution by age, gender, ethnicity, or other demographic factors, facilitating demographic studies and policy-making.

Code Snippet:

```
import matplotlib.pyplot as plt

labels = ['Company A', 'Company B', 'Company C', 'Company D']

sizes = [25, 35, 20, 20]   Market share percentages

colors = ['gold', 'yellowgreen', 'lightcoral', 'lightskyblue']

explode = (0.1, 0, 0, 0)   explode the 1st slice (i.e. 'Company A')

plt.pie(sizes, explode=explode, labels=labels, colors=colors,

        autopct='%1.1f%%', shadow=True, startangle=140)

plt.title('Market Share of Companies')
```

```
plt.axis('equal')

plt.show()
```



Description:

- **Data to Plot**: labels defines the names of the companies, sizes contains their respective market share percentages, colors assigns colors to each slice, and explode is used to "explode" or offset a specific slice for emphasis.
- **Creating the Pie Chart**: plt.pie generates the pie chart with the given data. autopct='%1.1f%%' displays the percentage value on each slice, shadow=True adds a shadow effect, and startangle=140 rotates the start angle for better visualization.
- **Adding a Title**: plt.title adds a title to the chart.
- **Displaying the Pie Chart**: plt.axis('equal') ensures the pie is drawn as a circle, and plt.show() renders the chart.

Box Plot

A box plot, also known as a whisker plot, is a graphical representation used to display the distribution, central tendency, and variability of a dataset. It shows the median, quartiles, and potential outliers, providing a summary of the data's spread and symmetry. The box represents the interquartile range (IQR) from the first quartile (Q1) to the third quartile (Q3), with a line at the median (Q2). Whiskers extend from the box to the smallest and largest values within 1.5 times the IQR from the quartiles. Outliers are typically plotted as individual points beyond the whiskers. Box plots are useful for comparing distributions across multiple groups and identifying skewness and outliers in the data.

Box plots are versatile tools for visual data analysis and are used in various fields to understand data distributions. Here are five use cases for box plots:

1. Comparative Analysis: Box plots are useful for comparing distributions across different groups or categories, such as comparing test scores across different schools or departments within an organization.

2. Identifying Outliers: Box plots help detect outliers in the data, which are values that fall significantly outside the expected range. This is valuable in quality control, anomaly detection, and data cleaning processes.

3. Understanding Data Distribution: They provide a clear summary of the data distribution, showing the median, quartiles, and overall spread, making it easy to understand the central tendency and variability of the data.

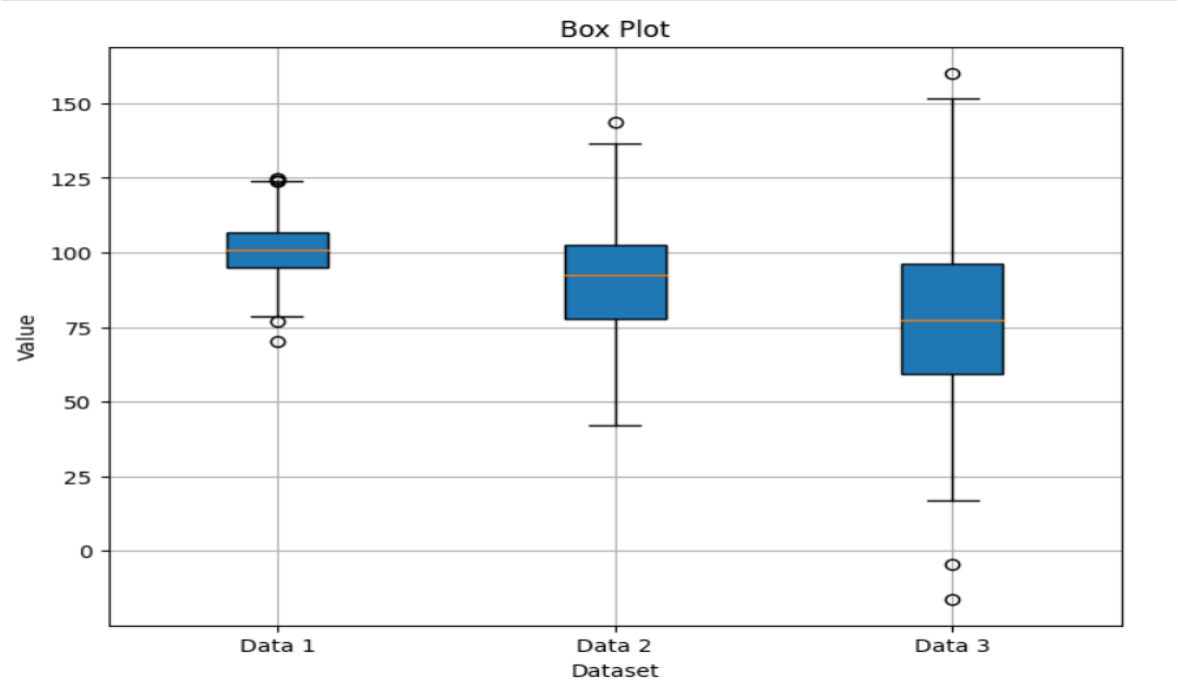
4. Clinical Research: In medical and clinical research, box plots can be used to compare the effects of different treatments or interventions on patient groups, helping to visualize differences in health outcomes.

5. Financial Analysis: Financial analysts use box plots to visualize the distribution of stock prices, returns, or other financial metrics over time, allowing them to assess volatility, identify trends, and compare different assets or portfolios.the data.

Code Snippet:

```
import matplotlib.pyplot as plt

import numpy as np

np.random.seed(10)

data1 = np.random.normal(100, 10, 200)

data2 = np.random.normal(90, 20, 200)

data3 = np.random.normal(80, 30, 200)

data = [data1, data2, data3]

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

plt.boxplot(data, patch_artist=True)
```

```
plt.title('Box Plot')
plt.xticks([1, 2, 3], ['Data 1', 'Data 2', 'Data 3'])
plt.xlabel('Dataset')
plt.ylabel('Value')
plt.grid(True)
plt.show()
```



Description:

1. **Import Libraries**: Import matplotlib.pyplot for plotting and numpy for generating random data.
2. **Generate Data**: Generate three sets of random data (data1, data2, data3) using numpy.random.normal.
3. **Combine Data**: Combine the data into a list (data).
4. **Create Box Plot**: Use plt.boxplot(data, patch_artist=True) to create the box plot. Setting patch_artist=True fills the boxes with colors.
5. **Customize Plot**: Add a title (plt.title), x-axis labels (plt.xticks), y-axis label (plt.ylabel), and enable grid (plt.grid(True)).
6. **Display Plot**: Use plt.show() to display the plot.

Area Plot

An area plot, also known as a stacked line plot or an area chart, displays quantitative data over time or categories using filled-in areas between lines. It is similar to a line plot but with the area under the line filled in with color, making it easier to visualize cumulative totals or proportions across different categories. Each category or dataset is represented by a colored area that starts from the x-axis and extends upward or downward, depending on the value at each point. Area plots are useful for illustrating trends and comparing contributions of different components to a whole, especially over time or across multiple categories.

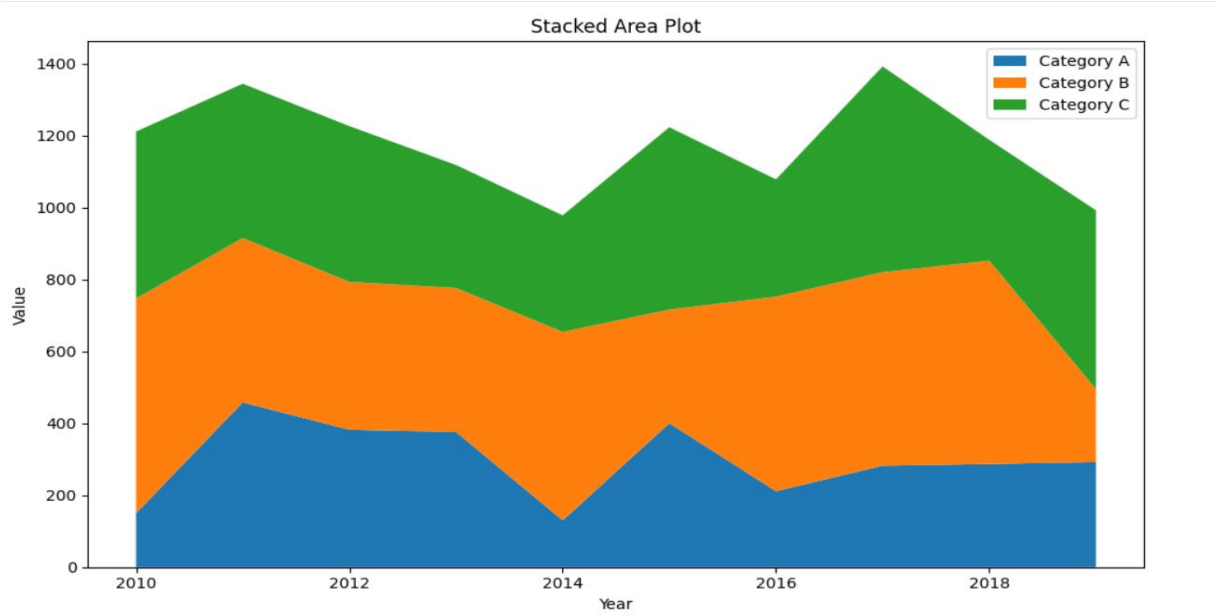
Area plots are versatile visualizations suitable for various data analysis scenarios. Here are five use cases where area plots can be particularly effective:

1. **Time-Series Analysis**: Visualize changes and trends over time for multiple variables or categories. For example, tracking quarterly sales performance of different product categories over several years.
2. **Stacked Proportions**: Compare the contribution of different components to a total or whole. For instance, showing the proportion of revenue contributed by different product lines each month.
3. **Distribution Comparison**: Illustrate how distributions vary across different groups or categories. For example, comparing the distribution of income levels across different regions or demographics.
4. **Cumulative Data**: Display cumulative sums or totals over time. This could include cumulative rainfall in different months of a year or cumulative spending in various budget categories.
5. **Hierarchical Data**: Represent hierarchical relationships where each layer contributes to the layer above it. This could involve visualizing how different expenditure categories contribute to total expenses over time.

Code Snippet:

```
import matplotlib.pyplot as plt

import numpy as np

years = np.arange(2010, 2020)

categories = ['Category A', 'Category B', 'Category C']

data = {

   'Category A': np.random.randint(100, 500, size=len(years)),

   'Category B': np.random.randint(200, 600, size=len(years)),

   'Category C': np.random.randint(300, 700, size=len(years))

}

plt.figure(figsize=(10, 6))
```

```
plt.stackplot(years, data.values(), labels=data.keys())
plt.title('Stacked Area Plot')
plt.xlabel('Year')
plt.ylabel('Value')
plt.legend()
plt.tight_layout()
plt.show()
```



Description:

1. **Import Libraries**: Import matplotlib.pyplot as plt and numpy as np.
2. **Generate Data**: Create random data for three categories (Category A, Category B, Category C) over a range of years (2010-2019).
3. **Create Area Plot**: Use plt.stackplot(years, data.values(), labels=data.keys()) to create the area plot. data.values() provides the numerical data for each category over the years, and data.keys() provides the labels for the legend.
4. **Customize Plot**: Add a title (plt.title), x-axis label (plt.xlabel), y-axis label (plt.ylabel), and legend (plt.legend()).
5. **Display Plot**: Use plt.tight_layout() to adjust the layout and plt.show() to display the plot.

Heatmap

A heatmap is a graphical representation of data where the individual values contained in a matrix are represented as colors. It is particularly useful for visualizing the magnitude of relationships or correlations between two sets of data. Typically, the matrix is displayed as a grid where each cell's color intensity corresponds to the value it represents. Heatmaps are commonly used in various fields such as statistics, data analysis, and biology to visualize patterns, correlations, or distributions in data. They provide a quick and intuitive way to identify trends and outliers in large datasets, making complex information more accessible and understandable at a glance.

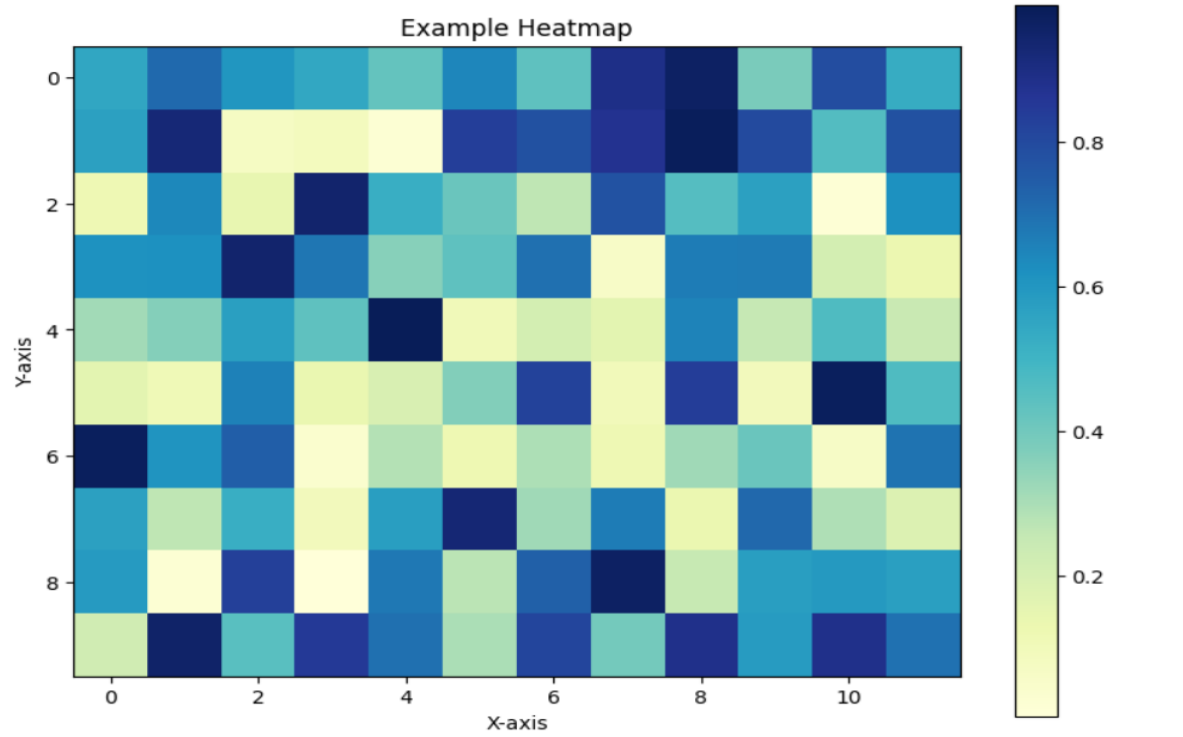
Heatmaps are versatile visualizations that find applications across different domains. Here are five common use cases where heatmaps are particularly valuable:

1. **Correlation Analysis**: Visualize correlations between variables in a dataset. Heatmaps use color intensity to represent the strength of correlation coefficients, helping to identify relationships and patterns among variables.
2. **Spatial Analysis**: Display geographical data such as population density, temperature variations, or economic indicators on maps. Heatmaps can depict data intensity across regions, providing insights into spatial patterns and distributions.
3. **User Behavior Analysis**: Analyze user interactions on websites or applications. Heatmaps can show where users click, scroll, or spend the most time, helping to optimize layouts, improve usability, and enhance user experience.
4. **Genomics and Biology**: Visualize gene expression levels, protein interactions, or biological pathways. Heatmaps are used to identify clusters of genes or proteins with similar expression patterns, aiding in biological research and medical diagnostics.
5. **Financial Analysis**: Monitor and analyze financial data such as stock prices, trading volumes, or market performance. Heatmaps can highlight trends, anomalies, or correlations across different financial instruments or market sectors, assisting in decision-making and risk management.

Code Snippet:

```
import matplotlib.pyplot as plt

import numpy as np

np.random.seed(0)

data = np.random.rand(10, 12)   Example 10x12 matrix of random values

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

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

plt.colorbar()

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

```
plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.tight_layout()

plt.show()
```



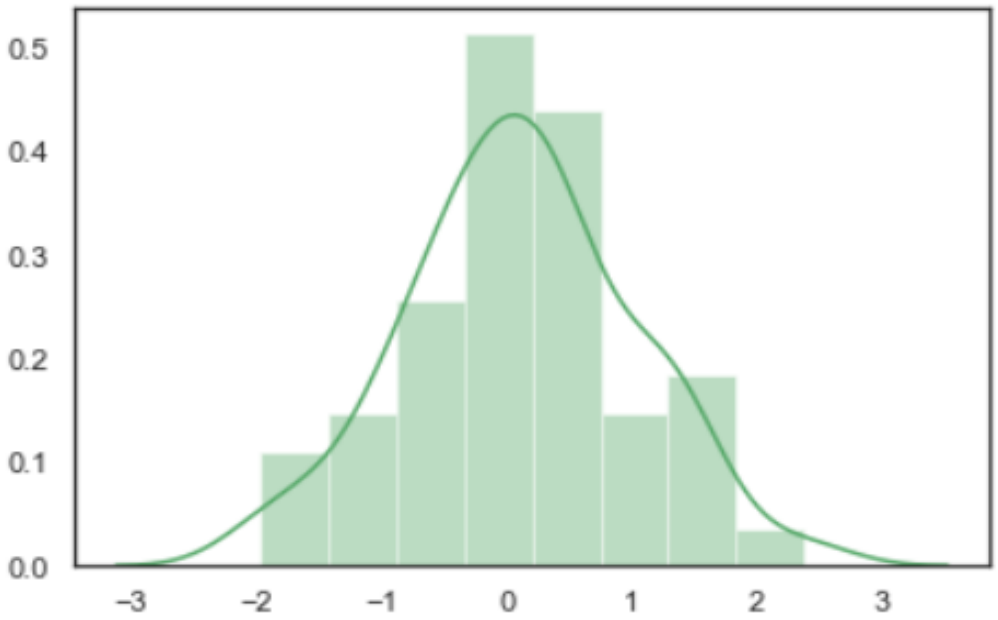
Description:

- **Import Libraries**: Import matplotlib.pyplot as plt and numpy as np.
- **Generate Data**: Create a 10x12 matrix of random values using np.random.rand() for demonstration purposes. Replace data with your actual dataset.
- **Create Heatmap**: Use plt.imshow(data, cmap='YlGnBu', interpolation='nearest') to create the heatmap:

- data is the matrix of values to be visualized.
- cmap='YlGnBu' sets the color palette (Yellow-Green-Blue in increasing order).
- interpolation='nearest' determines how the colors are interpolated between data points.
- **Add Colorbar**: Use plt.colorbar() to add a color bar indicating the scale of values.
- **Customize Plot**: Add a title (plt.title), x-axis label (plt.xlabel), and y-axis label (plt.ylabel) to the heatmap.
- **Display Plot**: Use plt.tight_layout() to adjust the layout and plt.show() to display the heatmap.

Seaborn

Seaborn is a powerful data visualization library for Python, built on top of Matplotlib, that simplifies the creation of complex and aesthetically pleasing statistical graphics. It integrates seamlessly with Pandas, allowing for easy data manipulation and plotting directly from DataFrames and Series. Seaborn offers a variety of high-level interfaces for different plot types, such as scatter plots, bar plots, and heatmaps, with minimal code. Its built-in themes enhance the visual appeal of plots, and it provides specialized functions for statistical visualizations like violin plots and pair plots. Additionally, Seaborn allows for extensive customization using Matplotlib's functionality, making it a versatile tool for data analysis and presentation.



Types of inputs to plotting functions:

Seaborn plotting functions can accept a variety of input types to create visualizations. Here are the primary types of inputs that Seaborn functions can take:

1. DataFrames

- **Pandas DataFrame**: Seaborn is designed to work seamlessly with Pandas DataFrames, making it easy to plot columns directly. Each column in the DataFrame can be referenced by its name in the plotting functions.

2. Lists

- **Python Lists**: Basic Python lists can also be used as inputs for Seaborn plots, providing a quick and easy way to visualize data.

Types of Plots in Seaborn:

1. **Pair Plot (pairplot)**: Creates a matrix of scatter plots for each pair of variables in the dataset, useful for exploring relationships.
2. **Swarm Plot (swarmplot)**: Displays each point in the data set with a non-overlapping distribution along one axis.
3. **Cat Plot (catplot)**: A general categorical plot that can create different types of categorical plots (strip, swarm, box, violin, boxen, point, bar, and count) using a common API.
4. **Joint Plot (jointplot)**: Combines a bivariate scatter plot or hexbin plot with univariate marginal distributions.
5. **Violin Plot (violinplot)**: Combines a box plot with a kernel density plot to show data distribution.
6. **Strip Plot (stripplot)**: A scatter plot where one of the variables is categorical, showing all observations along a categorical axis.
7. **Boxen Plot (boxenplot)**: An enhanced box plot for large datasets, emphasizing the distribution's tails.
8. **Facet Grid (FacetGrid)**: Maps a plot type (like scatter or line plot) across multiple subsets of data, creating a grid of plots.

Advantages:

Here are three key advantages of using Seaborn for data visualization:

1. Enhanced Visual Aesthetics:

Seaborn comes with built-in themes and color palettes that enhance the visual appeal of plots. It abstracts many of the low-level details required by Matplotlib, allowing users to create beautiful and informative graphics with minimal effort. This helps in making the plots not only more attractive but also more effective in conveying information.

2. Integration with Pandas:

Seaborn integrates seamlessly with Pandas DataFrames, allowing for easy and direct plotting of data stored in DataFrames and Series. This integration simplifies the workflow for data analysis and visualization, as users can manipulate data using Pandas and then create plots directly from the DataFrame without needing to convert it into another format.

3. Specialized Statistical Plots:

Seaborn is specifically designed for statistical data visualization, offering a wide range of specialized plot types such as violin plots, box plots, pair plots, and heatmaps. These plots are particularly useful for exploratory data analysis, helping users to quickly understand distributions, relationships, and patterns in the data. Seaborn also includes functions for automatically estimating and plotting statistical models, such as regression lines, which further enhance its utility for data analysis.

Disadvantages:

1. Limited Customization Compared to Matplotlib

Seaborn abstracts many of the low-level details of plot creation to simplify the process. While this is an advantage for ease of use, it can be a limitation for users who require fine-grained control over their plots. Matplotlib offers more detailed customization options, which can be necessary for highly specific visual requirements.

2. Dependency on Matplotlib

Seaborn is built on top of Matplotlib, meaning that it inherits some of its limitations and complexities. For advanced customizations, users often need to use Matplotlib functions, which can complicate the workflow and require additional knowledge of Matplotlib's API.

3. Performance with Large Datasets

Seaborn can be slower with large datasets compared to some other visualization libraries that are optimized for performance, such as Plotly or Bokeh. When working with very large datasets, users might experience longer rendering times and may need to resort to data sampling or alternative libraries for more efficient plotting.

Here are three to four scenarios when and why to use Seaborn for data visualization:

1. Exploratory Data Analysis (EDA)

When: When you are in the initial phase of analyzing a dataset and need to understand its structure, distribution, and relationships between variables.

Why: Seaborn provides a variety of plots (e.g., pair plots, joint plots, and distribution plots) that make it easy to explore data. Its integration with Pandas allows for quick and straightforward visualizations directly from DataFrames, helping you uncover patterns, trends, and outliers efficiently.

2. Statistical Visualization

When: When you need to visualize statistical relationships and summaries.

Why: Seaborn is designed with a focus on statistical data visualization. It offers specialized plots like violin plots, box plots, and heatmaps that can reveal insights into the data's distribution, variability, and correlation. This makes it ideal for visualizing statistical properties and comparisons.

4. Faceted and Multivariate Plots

When: When you need to visualize data across multiple subgroups or dimensions.

Why: Seaborn’s FacetGrid and catplot functions enable the creation of faceted plots, where you can display the same type of plot across multiple subsets of the data.

Pair Plot

A pair plot in the Seaborn library is a powerful visualization tool that allows you to explore relationships between multiple variables simultaneously. It generates a matrix of scatter plots for each pair of variables in the dataset, along with histograms or kernel density plots along the diagonal for each individual variable. This provides a comprehensive overview of the distribution and correlation between variables. Pair plots are particularly useful for identifying trends, clusters, and outliers, as well as understanding the structure of multi-dimensional data.

5 use cases of the Pair plot:

Multivariate Data Exploration: Pair plots are ideal for exploring the relationships between multiple variables in a dataset all at once. This is particularly useful when you have several variables and want to quickly understand how they interact with each other.

Pattern Recognition: Pair plots help in identifying patterns such as clusters or groups within the data. By visually inspecting the scatter plots, you can spot clusters of points that may correspond to distinct subgroups or classes in your data.

Correlation Analysis: Pair plots with correlation coefficients or trend lines provide insights into the strength and direction of relationships between variables. This is crucial for understanding which variables are positively, negatively, or weakly correlated with each other.

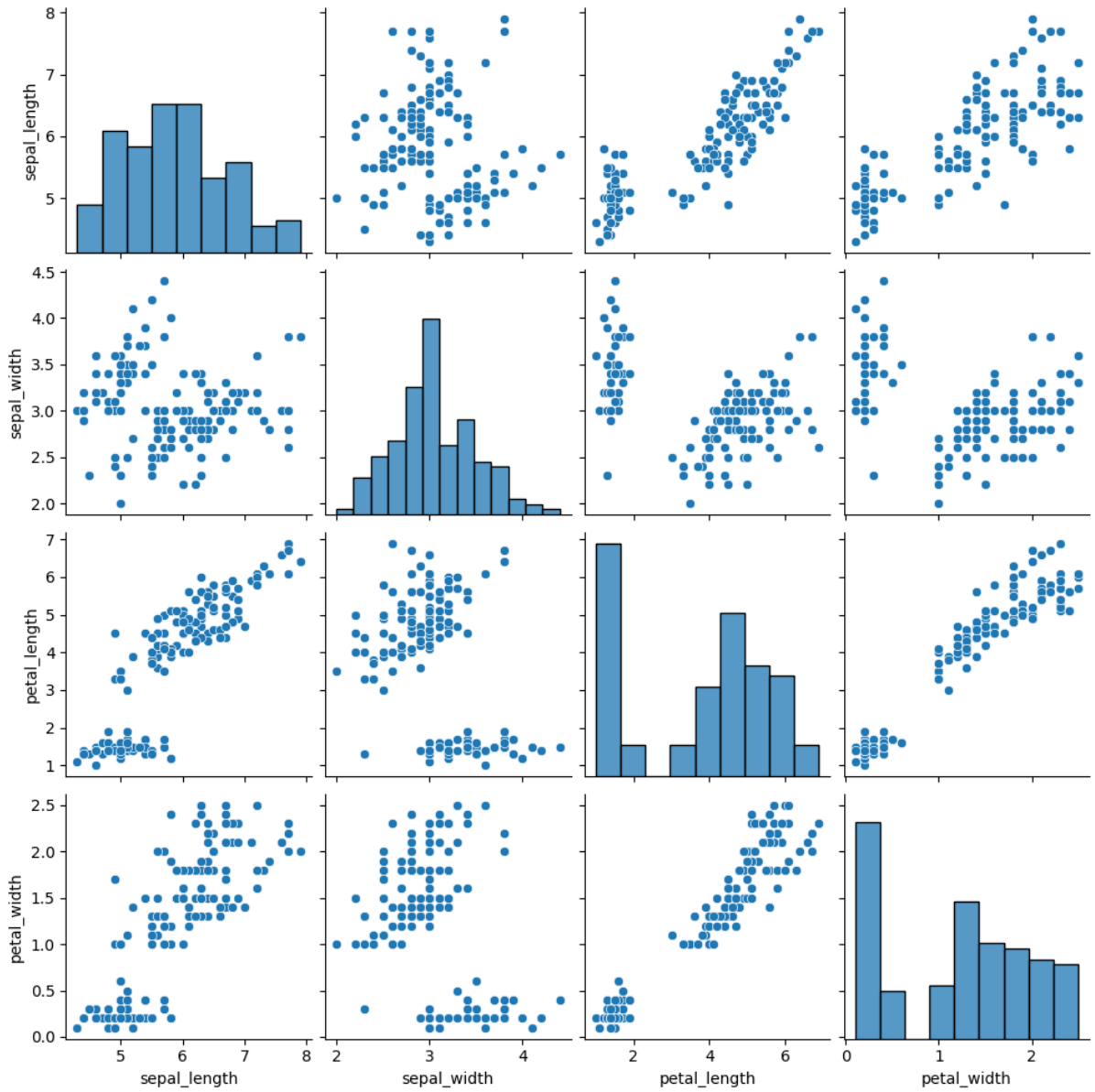
Outlier Detection: Outliers can be easily identified in pair plots as points that do not follow the general trend of the scatter plots. This helps in understanding if there are any unusual observations that may need further investigation.

Feature Selection: Pair plots can assist in feature selection by visually examining which variables are most likely to be informative predictors of the target variable. Variables that show strong relationships with the target or with each other may be more relevant for modeling.

Data Preprocessing Insights: Before applying machine learning algorithms, pair plots help in preprocessing steps such as normalization or scaling. They provide an understanding of the distribution of each variable and potential transformations that may be necessary.

Code Snippet:

```
import seaborn as sns

import matplotlib.pyplot as plt

import pandas as pd

iris = sns.load_dataset('iris')

sns.pairplot(iris)

plt.show()
```



Description:

- We import seaborn as sns and matplotlib.pyplot as plt.
- We load the Iris dataset using sns.load_dataset('iris'). You can replace 'iris' with any other dataset available in Seaborn or provide your own DataFrame.
- sns.pairplot(iris) generates a pair plot for all numerical columns in the Iris dataset. Each scatter plot shows the relationship between two variables, and the histograms show the distribution of each variable along the diagonal.
- plt.show() displays the pair plot.

Swarm plot

A swarm plot in the Seaborn library is a categorical scatter plot that displays individual data points along with their distribution. It arranges each data point in a way that avoids overlap, making it particularly useful for visualizing categorical data with many levels. Swarm plots provide insights into the distribution and density of data points within each category, showing where values are concentrated and how they vary. They are effective for comparing distributions between categories and are often used in exploratory data analysis to understand the spread and pattern of categorical variables across different groups or classes.

Here are five use cases where swarm plots from the Seaborn library can be effectively utilized:

1. Comparison of Categorical Variables: Swarm plots are ideal for comparing distributions of a numerical variable across different categories. For instance, you can visualize how the distribution of exam scores varies across different school grades or how customer ratings differ between various product categories.

2. Identifying Outliers: Swarm plots help in identifying outliers within each category by displaying individual data points. Outliers can be easily spotted as points that deviate significantly from the general distribution of data within a category.

3. Interaction Effects: Swarm plots can reveal interaction effects between categorical variables. For example, you can explore how the relationship between two variables (e.g., age and income) varies across different demographic groups (e.g., gender or education level).

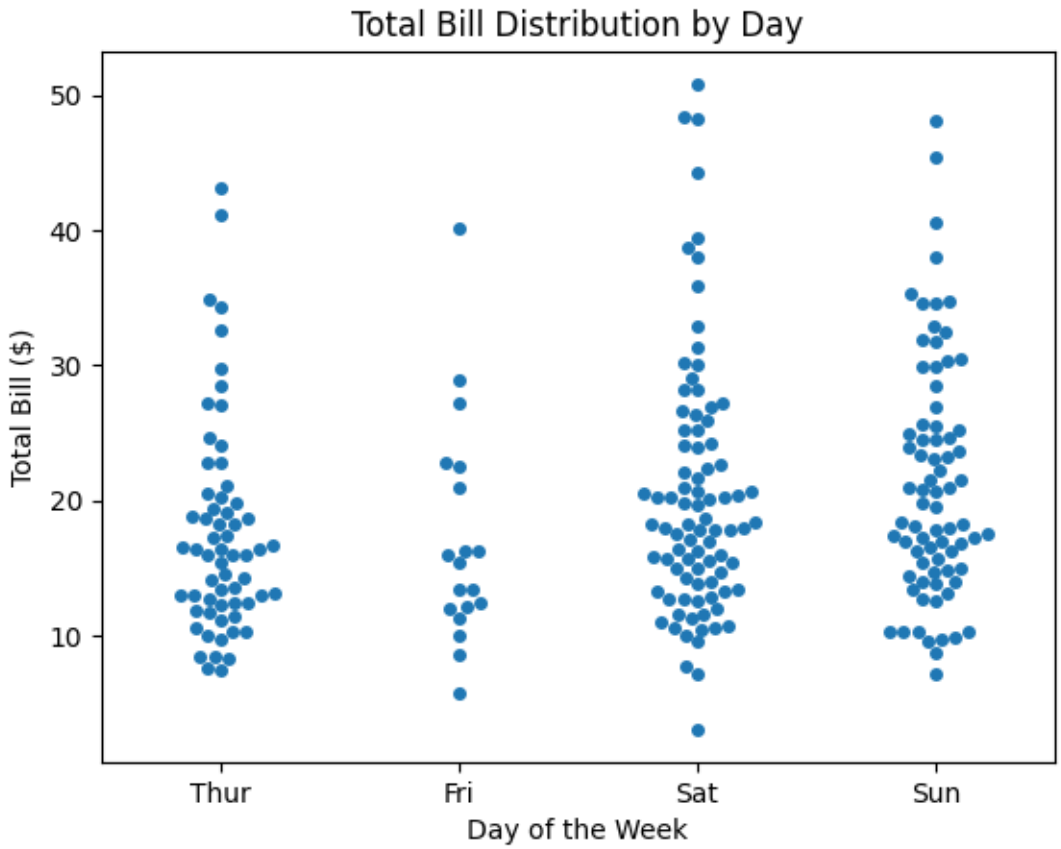
4. Data Density Visualization: Swarm plots provide a clear visual representation of data density within each category. Dense regions indicate where most data points are concentrated, while sparse regions indicate areas with fewer observations.

5. Pattern Recognition: Swarm plots are useful for detecting patterns or trends within categorical data. They can reveal clustering or grouping of data points within each category, which may suggest underlying structures or relationships in the data.

Code Snippet:

```
import seaborn as sns

import matplotlib.pyplot as plt

tips = sns.load_dataset('tips')

sns.swarmplot(x='day', y='total_bill', data=tips)

plt.xlabel('Day of the Week')

plt.ylabel('Total Bill ($)')

plt.title('Total Bill Distribution by Day')
```

plt.show()



Description:

- We import seaborn as sns and matplotlib.pyplot as plt.
- We load the 'tips' dataset using sns.load_dataset('tips'), which is a built-in dataset in Seaborn containing information about tips in a restaurant.
- sns.swarmplot() is used to create the swarm plot. We specify x='day' and y='total_bill' to plot the total bill amount on the y-axis for each day of the week on the x-axis.
- plt.xlabel(), plt.ylabel(), and plt.title() are used to add labels and a title to the plot.
- plt.show() displays the swarm plot.

Cat Plot

A cat plot (short for categorical plot) in Seaborn is a versatile visualization tool that allows for the simultaneous display of several categorical plots. It encompasses a variety of categorical plot types, including strip plots, swarm plots, box plots, violin plots, and more, making it useful for comparing distributions of numerical variables across different categories. Cat plots are customizable and can be tailored to show additional dimensions of data, such as hue for grouping by a third categorical variable or col/row for displaying facets of data across multiple subplots. They are widely used in exploratory data analysis and statistical visualization to uncover patterns, trends, and relationships within categorical data sets.

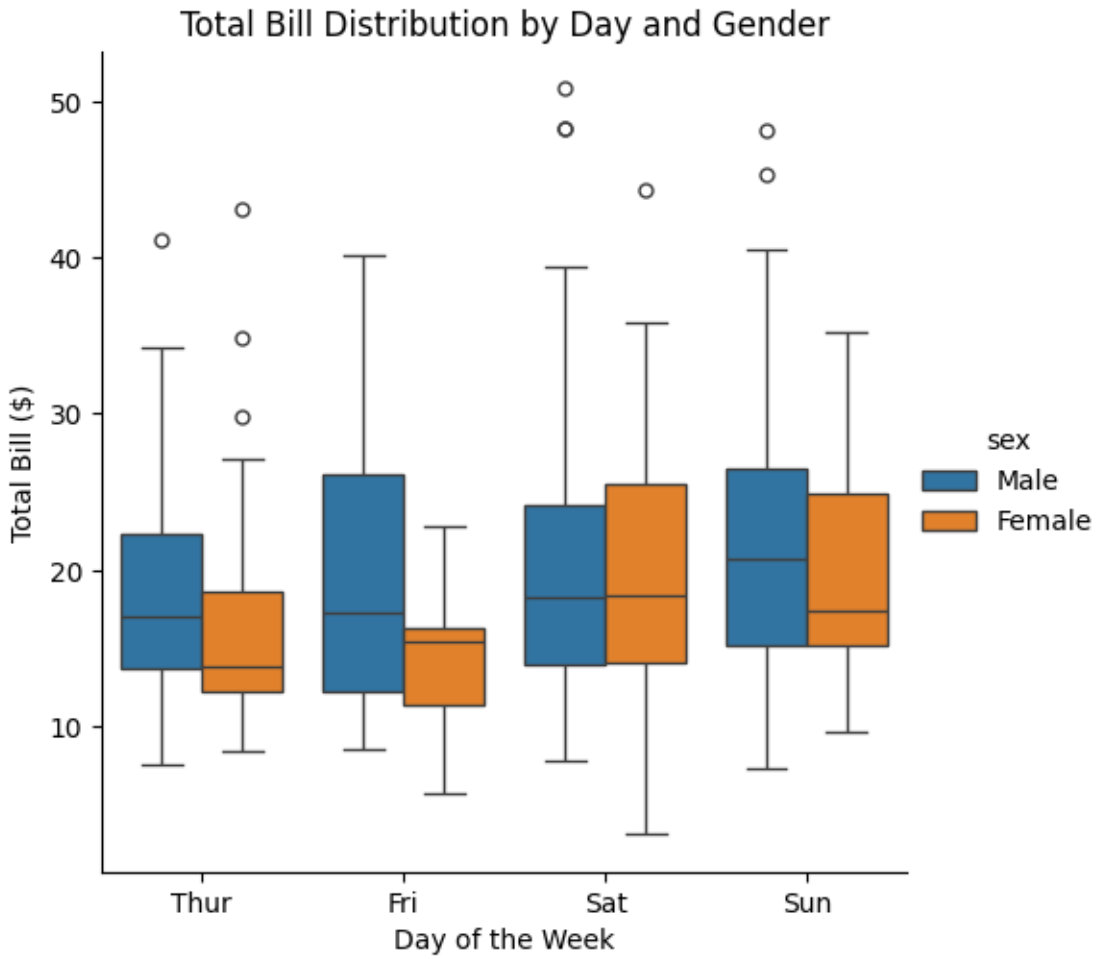
Here are 5 use cases where cat plots (categorical plots) from Seaborn can be effectively utilized:

1. **Comparing Distributions**: Cat plots are ideal for comparing distributions of numerical variables across different categories. For example, you can visualize how salaries vary across different job titles or how customer ratings differ between various product categories using box plots or violin plots.
2. **Visualizing Trends Over Time**: Cat plots can be used to visualize trends over time or across different groups. For instance, you can plot the average sales volume per month over several years using a point plot, allowing you to identify seasonal patterns or trends.
3. **Exploring Interaction Effects**: Cat plots help in exploring interaction effects between categorical variables. For example, you can visualize how the relationship between customer satisfaction and purchase intent varies across different demographic groups using point plots with hue encoding.
4. **Detecting Outliers and Anomalies**: Cat plots are useful for detecting outliers and anomalies within each category. By visualizing individual data points (e.g., using swarm plots), you can identify observations that deviate significantly from the typical distribution within a category.
5. **Comparing Models or Algorithms**: In machine learning and data science, cat plots can be used to compare the performance metrics (e.g., accuracy, F1 score) of different models or algorithms across different datasets or experimental conditions, providing insights into their relative strengths and weaknesses.

Code Snippet:

```
import seaborn as sns

import matplotlib.pyplot as plt

tips = sns.load_dataset('tips')

sns.catplot(x='day', y='total_bill', hue='sex', kind='box', data=tips)

plt.xlabel('Day of the Week')

plt.ylabel('Total Bill ($)')

plt.title('Total Bill Distribution by Day and Gender')
```

plt.show()



Description:

- We import seaborn as sns and matplotlib.pyplot as plt.
- We load the 'tips' dataset using sns.load_dataset('tips'), which is a built-in dataset in Seaborn containing information about tips in a restaurant.
- sns.catplot() is used to create the cat plot. We specify x='day' and y='total_bill' to plot the total bill amount on the y-axis for each day of the week on the x-axis. The hue='sex' parameter splits the data by gender, and kind='box' specifies that we want to use a box plot for visualization.
- plt.xlabel(), plt.ylabel(), and plt.title() are used to add labels and a title to the plot.
- plt.show() displays the cat plot.

Joint Plot

A joint plot in Seaborn is a versatile visualization that combines multiple plots to showcase the relationship between two variables. It typically includes a scatter plot for visualizing the joint distribution of the variables, histograms or kernel density estimation (KDE) plots along the axes to represent the marginal distributions of each variable separately, and optionally a correlation coefficient. This makes joint plots useful for exploring correlations, trends, and distributions within bivariate data. They offer insights into how variables interact and are distributed, aiding in both exploratory data analysis and hypothesis-testing tasks.

Here are five use cases where joint plots from Seaborn can be effectively utilized:

1. Exploring Correlation: Joint plots are ideal for examining the correlation between two variables. By visualizing the scatter plot along with the correlation coefficient, analysts can quickly assess whether variables are positively, negatively, or weakly correlated.

2. Identifying Trends: Joint plots help in identifying trends and patterns within bivariate data. For example, they can reveal whether there is a linear relationship between variables or if there are clusters of data points that suggest non-linear associations.

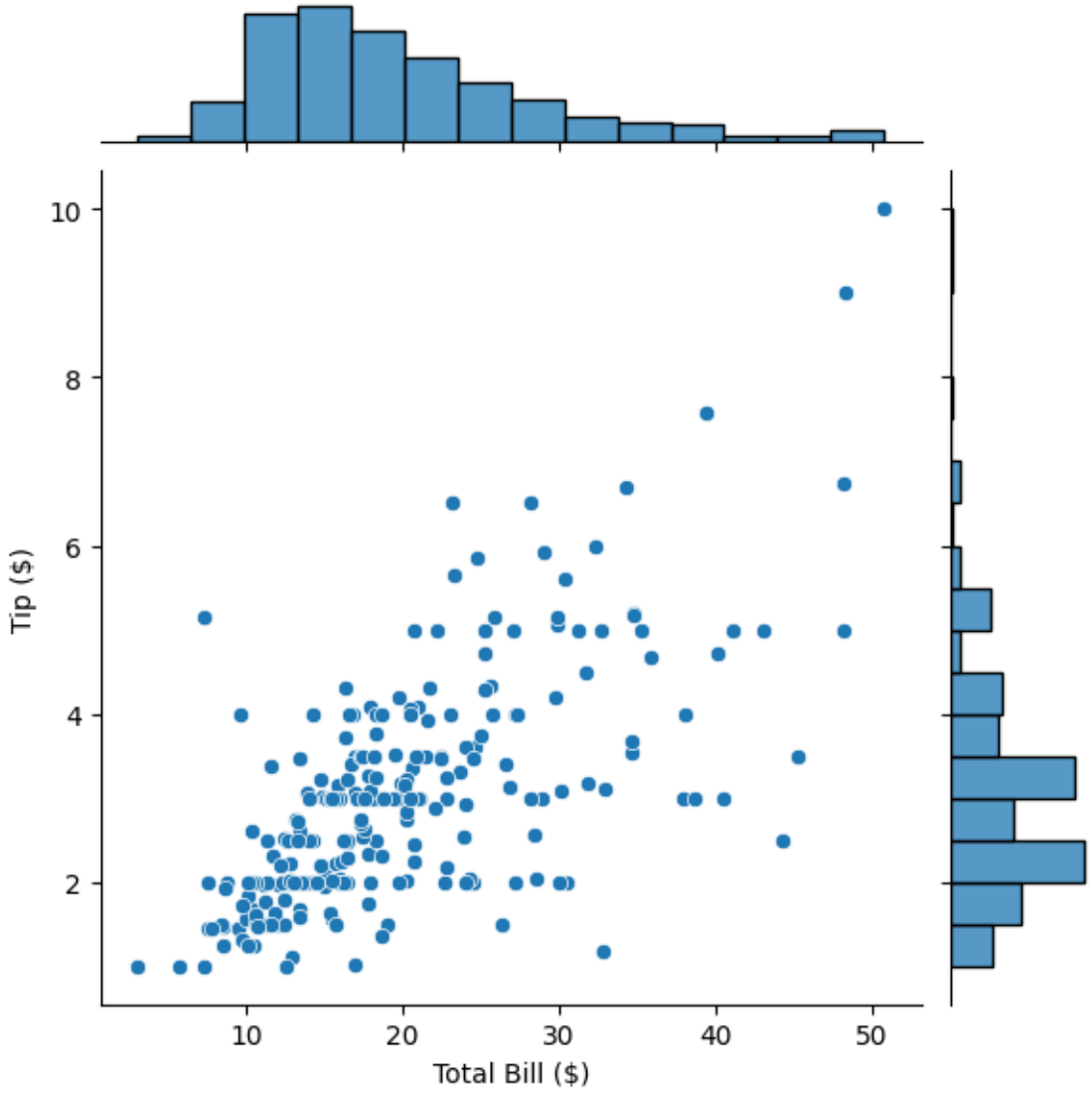
3. Comparing Distributions: Joint plots facilitate the comparison of distributions between two variables. Histograms or KDE plots along the axes provide insights into the marginal distributions of each variable, showcasing their spread and skewness.

4. Outlier Detection: Joint plots can aid in outlier detection by highlighting data points that deviate significantly from the general pattern observed in the scatter plot. Outliers can indicate anomalies or unusual observations that may require further investigation.

5. Model Validation: Joint plots are useful for validating assumptions made during model development. For instance, they can be used to check if the relationship between predictors and the target variable aligns with expectations or if there are systematic deviations indicating model misfit.

Code Snippet:

```
import seaborn as sns

import matplotlib.pyplot as plt

tips = sns.load_dataset('tips')

sns.jointplot(x='total_bill', y='tip', data=tips, kind='scatter')

plt.xlabel('Total Bill ($)')

plt.ylabel('Tip ($)')

plt.show()
```



Description:

- We import seaborn as sns and matplotlib.pyplot as plt.
- We load the 'tips' dataset using sns.load_dataset('tips'), which is a built-in dataset in Seaborn containing information about tips in a restaurant.
- sns.jointplot() is used to create the joint plot. We specify x='total_bill' and y='tip' to plot the relationship between total bill amount (x-axis) and tip amount (y-axis). The kind='scatter' parameter specifies that we want to visualize this relationship using a scatter plot.
- plt.xlabel(), plt.ylabel(), and plt.title() are used to add labels and a title to the plot.
- plt.show() displays the joint plot.

Violin Plot

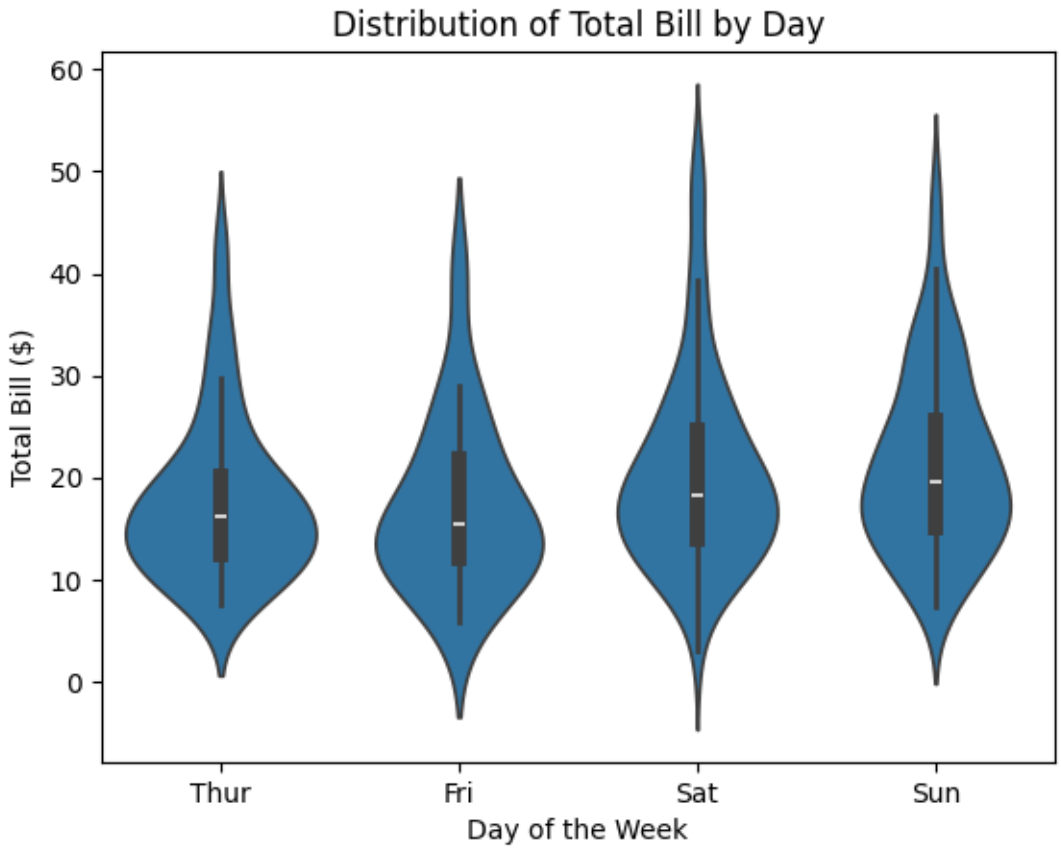
A violin plot in Seaborn is a statistical visualization that combines aspects of a box plot and a kernel density plot. It displays the distribution of quantitative data across several levels of one or more categorical variables. The width of the violin indicates the density or frequency of data points at different values, while the inner part shows the probability density of the data at each level. Violin plots are useful for comparing distributions and detecting differences in variability or skewness between groups. They provide a comprehensive view of data distribution and are particularly effective for visualizing complex datasets with multiple categories or levels.

Here are five use cases where violin plots from Seaborn can be effectively utilized:

1. **Comparing Distribution Shapes**: Violin plots are ideal for comparing the shapes of distributions across different categories or groups. For example, you can visualize the distribution of exam scores across different school grades to see if there are differences in the spread or skewness of scores.
2. **Identifying Outliers and Skewness**: Violin plots help in identifying outliers and understanding the skewness of data within each category. Outliers appear as points outside the violin plot's shape, indicating extreme values that may require further investigation.
3. **Visualizing Multimodal Distributions**: Violin plots are effective for visualizing multimodal distributions, where data may exhibit multiple peaks or modes within each category. This helps in understanding complex patterns and distributions that may not be apparent with simpler plots.
4. **Comparing Group Statistics**: Violin plots can display summary statistics such as median, quartiles, and range within each category, making it easy to compare central tendencies and variability between groups or categories.
5. **Interaction Effects**: Violin plots are useful for visualizing interaction effects between categorical variables. For instance, you can explore how the distribution of income varies across different education levels and genders simultaneously, revealing potential interactions between these variables.

Code Snippet:

```
import seaborn as sns

import matplotlib.pyplot as plt

tips = sns.load_dataset('tips')

sns.violinplot(x='day', y='total_bill', data=tips)

plt.xlabel('Day of the Week')

plt.ylabel('Total Bill ($)')

plt.title('Distribution of Total Bill by Day')

plt.show()
```



Description:

- We import seaborn as sns and matplotlib.pyplot as plt.
- We load the 'tips' dataset using sns.load_dataset('tips'), which is a built-in dataset in Seaborn containing information about tips in a restaurant.
- sns.violinplot() is used to create the violin plot. We specify x='day' and y='total_bill' to visualize the distribution of total bill amounts across different days of the week.
- plt.xlabel(), plt.ylabel(), and plt.title() are used to add labels and a title to the plot.
- plt.show() displays the violin plot.

Strip Plot

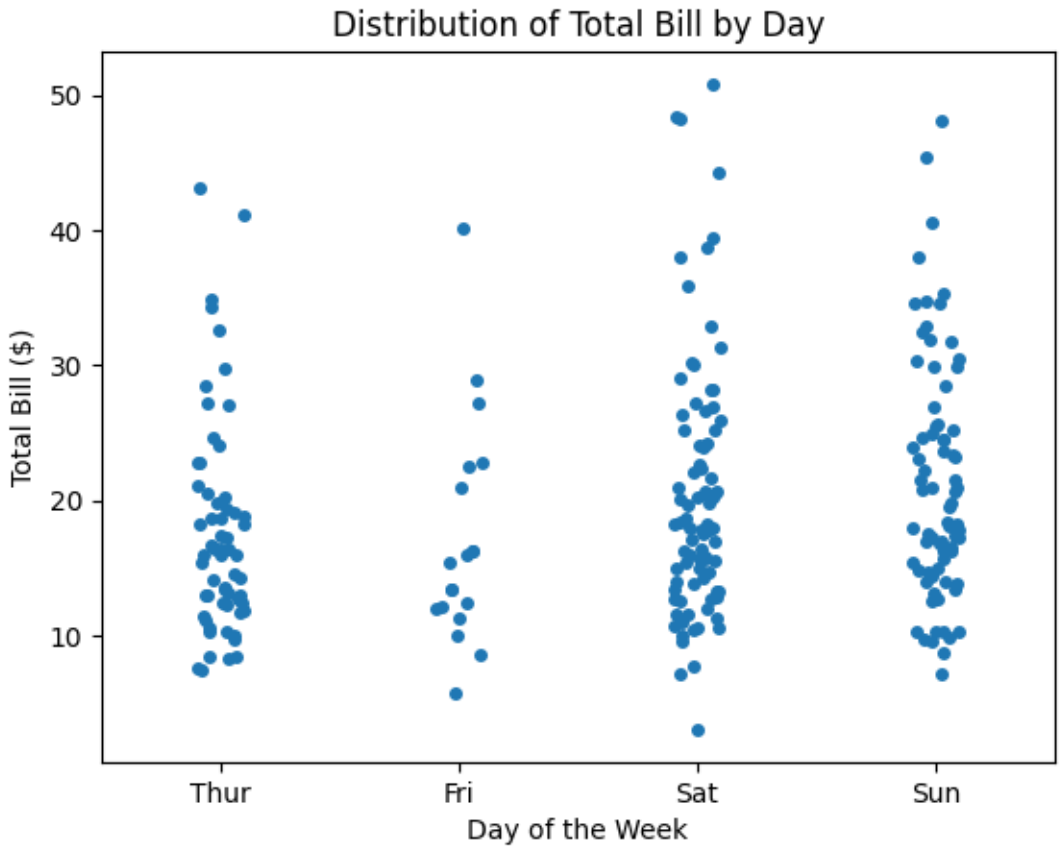
A strip plot in Seaborn is a type of categorical plot that displays individual data points as dots along an axis, typically representing one categorical variable. It is useful for visualizing the distribution and density of data points within each category. Strip plots are straightforward yet effective for identifying patterns, outliers, and the overall distribution of numerical data across different levels of a categorical variable. They provide a clear view of the spread and concentration of data points, making them suitable for exploratory data analysis and comparison between categories.

Here are five use cases where strip plots from Seaborn can be effectively utilized:

1. **Comparison of Continuous Variables**: Strip plots are ideal for comparing distributions of a continuous variable across different categories. For example, you can visualize how the distribution of student grades varies across different subjects or courses.
2. **Detection of Outliers**: Strip plots help in identifying outliers within each category by displaying individual data points as dots. Outliers can be easily spotted as points that deviate significantly from the general distribution of data within a category.
3. **Exploring Relationships**: Strip plots can reveal relationships between categorical and continuous variables. For instance, you can investigate how the distribution of house prices varies across different neighborhoods or regions.
4. **Pattern Recognition**: Strip plots are useful for detecting patterns or trends within categorical data. They can highlight clusters or groups of data points within each category, indicating potential patterns or associations.
5. **Data Density Visualization**: Strip plots provide a visual representation of data density within each category. They show where data points are concentrated or sparse, offering insights into the distribution and variability of the data.

Code Snippet:

```
import seaborn as sns

import matplotlib.pyplot as plt

tips = sns.load_dataset('tips')

sns.stripplot(x='day', y='total_bill', data=tips)

plt.xlabel('Day of the Week')

plt.ylabel('Total Bill ($)')

plt.title('Distribution of Total Bill by Day')

plt.show()
```



Description:

- We import seaborn as sns and matplotlib.pyplot as plt.
- We load the 'tips' dataset using sns.load_dataset('tips'), which is a built-in dataset in Seaborn containing information about tips in a restaurant.
- sns.stripplot() is used to create the strip plot. We specify x='day' and y='total_bill' to visualize the distribution of total bill amounts across different days of the week.
- plt.xlabel(), plt.ylabel(), and plt.title() are used to add labels and a title to the plot.
- plt.show() displays the strip plot.

Boxen Plot

A Boxen plot, also known as a letter-value plot or a box plot with enhanced representation, is an enhanced version of the traditional box plot in Seaborn. It displays the distribution of quantitative data with added emphasis on higher quantiles, making it particularly useful for datasets with large numbers of observations and varying scales. The plot includes additional quartile lines (hence the name "boxen") that provide more detailed insights into the data distribution. Boxen plots are effective for detecting outliers, comparing groups, and visualizing the spread and skewness of data across different categories or groups. They are valuable tools in exploratory data analysis and statistical visualization tasks.

Here are five use cases where Boxen plots from Seaborn can be effectively utilized:

1. Handling Large Datasets: Boxen plots are particularly useful for visualizing large datasets with many observations. They provide a clearer representation of the distribution of data compared to traditional box plots by showing more quantile lines and allowing for better interpretation of data density and spread.

2. Comparing Multiple Groups: Boxen plots are effective for comparing the distribution of a numerical variable across multiple categories or groups. For example, you can visualize how income levels vary across different education levels or job categories, providing insights into socioeconomic patterns.

3. Identifying Outliers: Boxen plots help in identifying outliers within each category or group. The additional quartile lines provide a detailed view of the data distribution, making it easier to detect extreme values that may indicate anomalies or interesting observations.

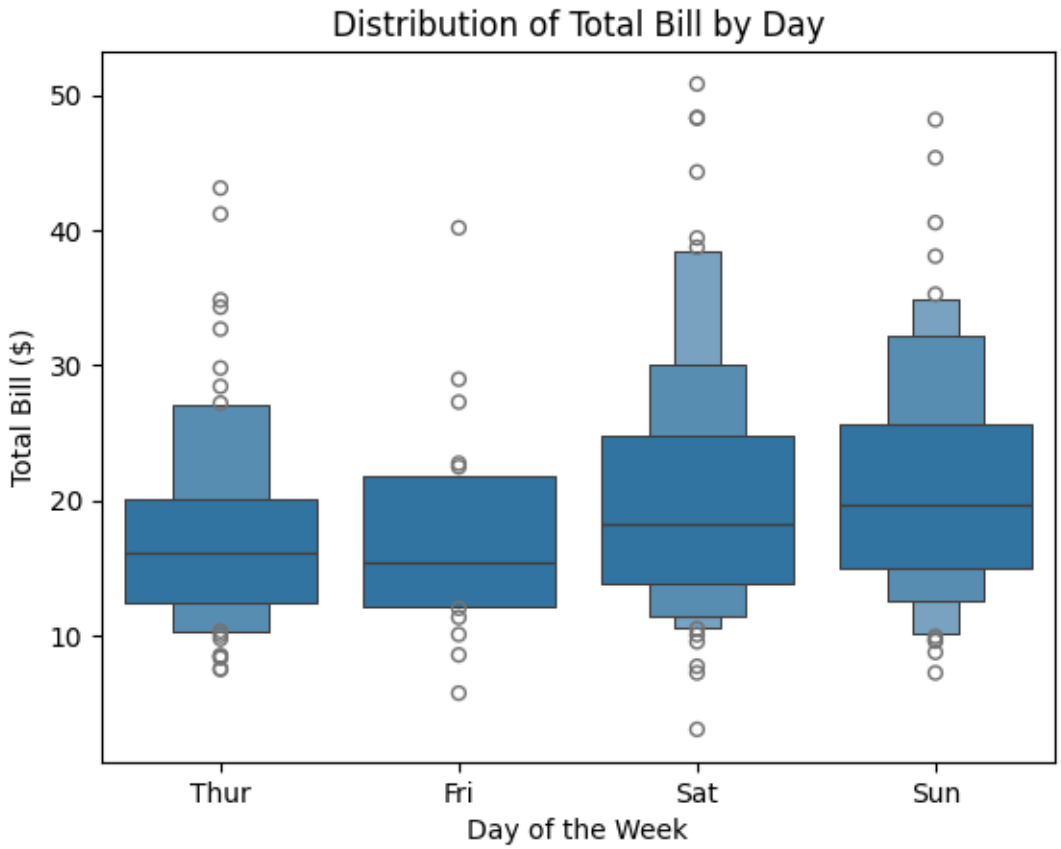
4. Detecting Skewness and Variability: Boxen plots are useful for detecting skewness and variability in data distributions across categories. They allow for a comprehensive assessment of the spread and shape of data, revealing whether distributions are symmetric or skewed within each group.

5. Comparison with Traditional Box Plots: Boxen plots can be used to compare distributions with traditional box plots. They offer a more detailed view of the data distribution at the expense of a slightly more complex visualization, making them suitable for situations where a more granular understanding of data distribution is needed.

Code Snippet:

```
import seaborn as sns

import matplotlib.pyplot as plt

tips = sns.load_dataset('tips')

sns.boxenplot(x='day', y='total_bill', data=tips)

plt.xlabel('Day of the Week')
```

```
plt.ylabel('Total Bill ($)')

plt.title('Distribution of Total Bill by Day')

plt.show()
```



Description:

- We import seaborn as sns and matplotlib.pyplot as plt.
- We load the 'tips' dataset using sns.load_dataset('tips'), which is a built-in dataset in Seaborn containing information about tips in a restaurant.
- sns.boxenplot() is used to create the Boxen plot. We specify x='day' and y='total_bill' to visualize the distribution of total bill amounts across different days of the week.
- plt.xlabel(), plt.ylabel(), and plt.title() are used to add labels and a title to the plot.
- plt.show() displays the Boxen plot.

Facet Grid

FacetGrid in Seaborn is a powerful tool for creating multiple plots arranged in a grid based on one or more categorical variables. It allows you to visualize relationships between variables across multiple levels of categories simultaneously. Each subplot in the grid represents a subset of the data defined by unique combinations of categorical variables. FacetGrid facilitates comparison and exploration of data patterns across different facets, making it effective for complex data visualization tasks and understanding interactions between variables within categorical groups.

Here are five use cases where FacetGrid from Seaborn can be effectively utilized:

1. Comparison Across Categories: FacetGrid is ideal for comparing distributions or relationships across multiple categories. For example, you can create separate plots for different product categories to compare sales trends or customer preferences.

2. Exploring Relationships: FacetGrid allows you to visualize relationships between variables within different subsets of data. You can plot scatter plots, regressions, or other visualizations to explore how the relationship between variables varies across different groups.

3. Time Series Analysis: FacetGrid can be used to analyze time series data by grouping plots based on different time periods, seasons, or other temporal categories. This helps in visualizing trends, seasonality, and patterns over time.

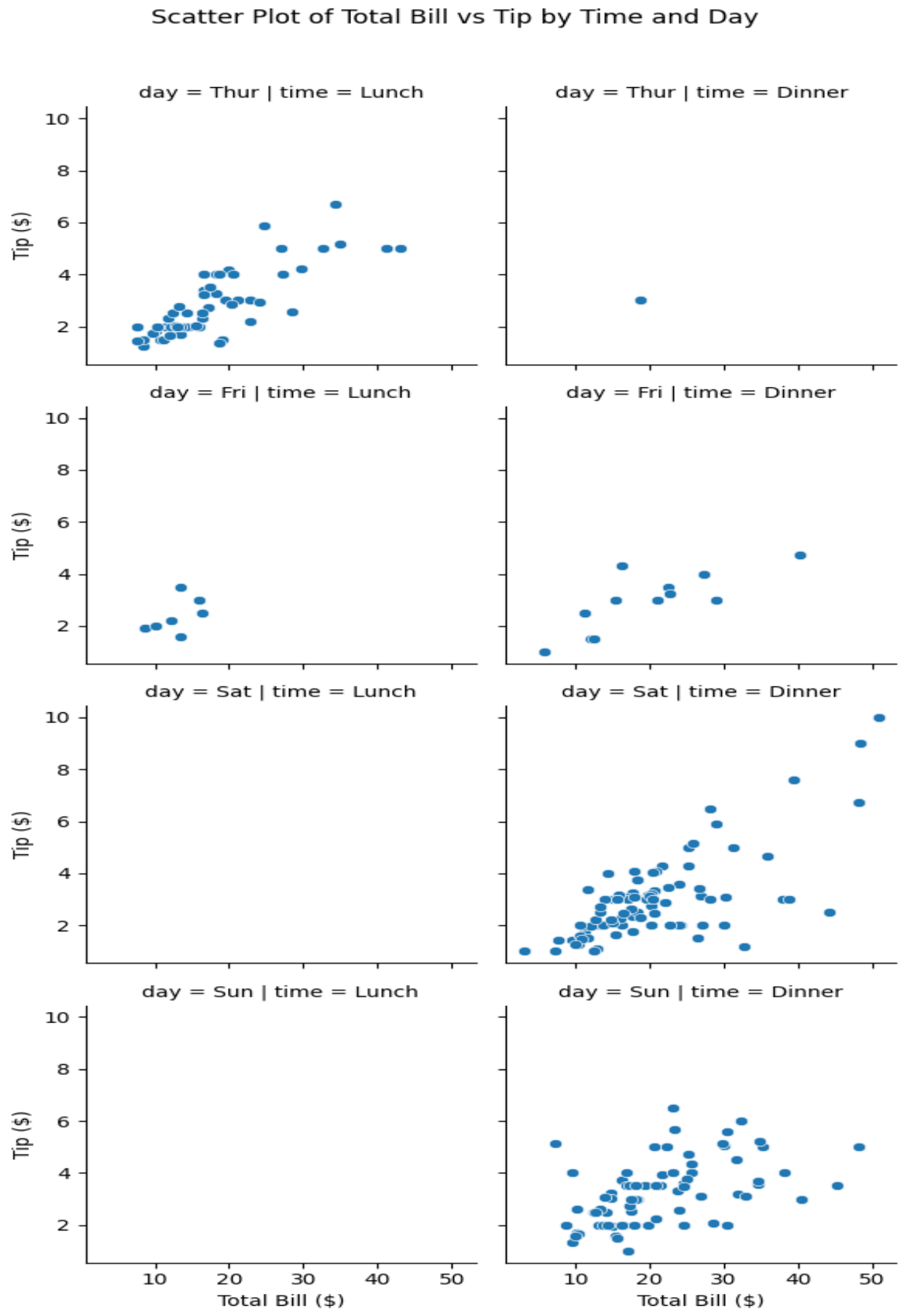
4. Multi-dimensional Analysis: FacetGrid facilitates multi-dimensional analysis by allowing you to add more dimensions to your visualization through row and column facets. This is useful for exploring interactions between multiple variables simultaneously.

5. Comparing Model Performance: In machine learning and data science, FacetGrid can be used to compare the performance metrics (e.g., accuracy, F1 score) of different models across different datasets or experimental conditions. Each facet can represent a different model or dataset, providing insights into model effectiveness and variability.

Code Snippet:

```
import seaborn as sns

import matplotlib.pyplot as plt

tips = sns.load_dataset('tips')

g = sns.FacetGrid(tips, col='time', row='day')

g.map(sns.scatterplot, 'total_bill', 'tip')

g.set_axis_labels('Total Bill ($)', 'Tip ($)')

g.fig.suptitle('Scatter Plot of Total Bill vs Tip by Time and Day', y=1.02)
```

```
plt.tight_layout()

plt.show()
```



Description:

- We import seaborn as sns and matplotlib.pyplot as plt.
- We load the 'tips' dataset using sns.load_dataset('tips'), which is a built-in dataset in Seaborn containing information about tips in a restaurant.
- sns.FacetGrid() initializes a FacetGrid object with the tips DataFrame. We specify col='time' and row='day', which means we want to create separate plots for each unique combination of 'time' (lunch or dinner) and 'day' (Thursday, Friday, Saturday, or Sunday).
- g.map(sns.scatterplot, 'total_bill', 'tip') specifies that we want to create scatter plots of 'total_bill' versus 'tip' for each facet.
- g.set_axis_labels() sets the labels for the x-axis and y-axis for each plot.
- g.fig.suptitle() adds a title to the entire FacetGrid.

This code snippet will produce a FacetGrid with scatter plots showing the relationship between 'total_bill' and 'tip' for different times of day (lunch vs dinner) across different days of the week (Thursday to Sunday).