Data Analytics - Module II: Data Visualization with Python

Objectives

- Learn to visualize data with Matplotlib through line plots, scatter plots, bar charts, histograms, and box plots.
- Understand the creation of heatmaps using Seaborn to visualize correlation matrices.
- Gain proficiency in Seaborn's regression plots for analyzing linear relationships between variables.
- Master the creation of scatter plots with Matplotlib to display individual data points.
- Acquire skills in generating bar charts with Matplotlib to compare data across categories.

Introduction

In this session, we will delve into the art of data visualization using Matplotlib and Seaborn, two powerful Python libraries. Visualization is a crucial skill for anyone working with data, as it enables us to transform complex datasets into clear and informative visual representations, such as charts, graphs, and plots. By learning visualization, you gain the ability to uncover patterns, trends, and insights within your data, making it easier to communicate findings and make data-driven decisions.

Matplotlib and Seaborn offer a wide range of customization options, making them invaluable tools for creating visually appealing and meaningful visualizations that can enhance data understanding, aid in storytelling, and facilitate effective data-driven communication. Whether you are a data scientist, analyst, or anyone working with data, mastering these visualization libraries is a fundamental step in your journey toward becoming a proficient data professional.

Note

To install Matplotlib and Seaborn using pip, you can activate your virtual environment, open your command prompt or terminal and use the following commands

- pip install matplotlib seaborn

1. Matplotlib

Let's go through each visualization example step by step, starting with an explanation of what we are going to do, followed by the example code, and then a detailed explanation of the code.

1.1. Line Plot

Line plots are used to visualize trends and changes in data over a continuous range, or time. We use line plots when we have data that can be represented as a series of points connected by lines, such as time series data or data with a natural ordering.

We will create a simple line plot using Matplotlib to visualize a set of data points.

Code:

```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 12, 5, 8, 7]

# Create a basic line plot
plt.plot(x, y)

# Add labels and a title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Line Plot Example')
```

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

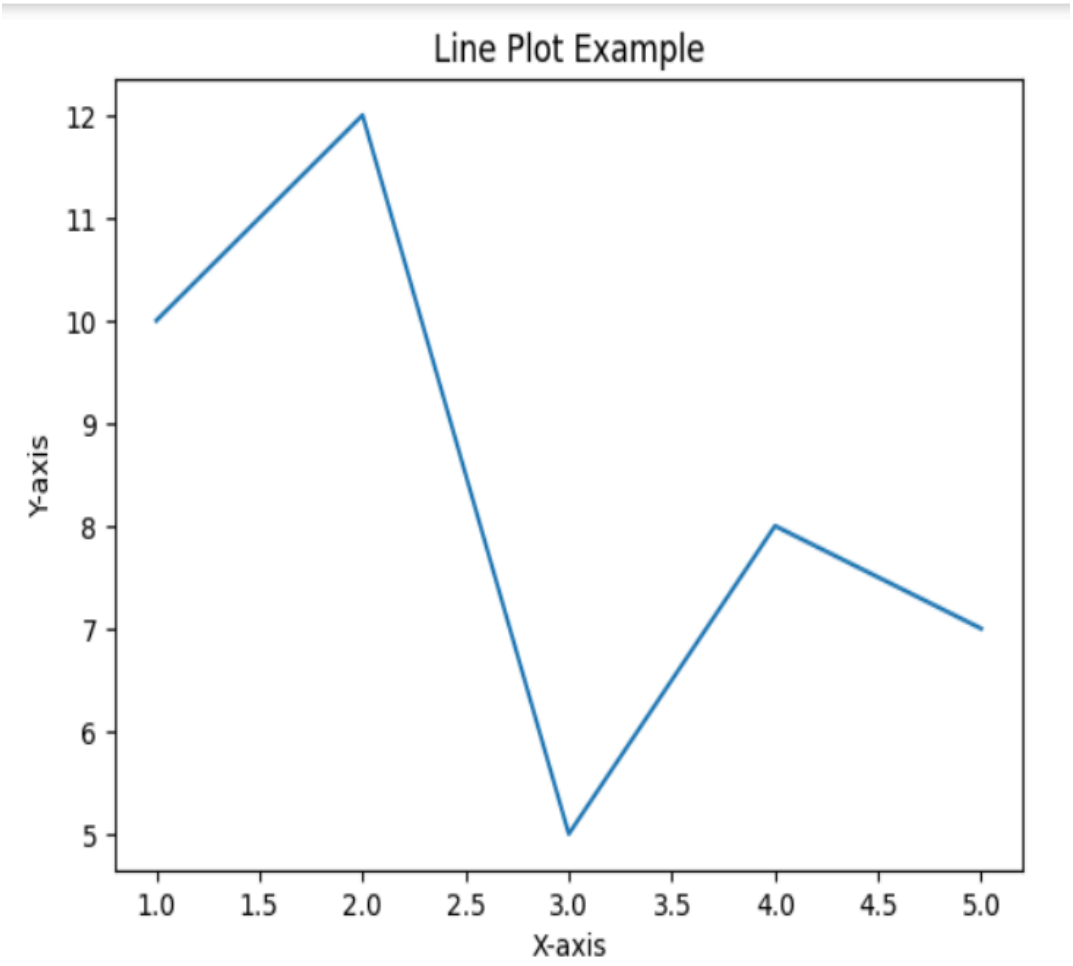

Figure 1: Line Plot

The line plot suggests a fluctuating trend in the data. There's an initial increase in ***y*** values, followed by a sudden drop, then a slight recovery towards the end. This indicates some kind of variation or change in the underlying phenomenon being represented.

Code Explanation

- ***import matplotlib.pyplot as plt***: This line imports the Matplotlib library, specifically the pyplot module, and aliases it as plt, which is a common convention.
- ***x*** and ***y*** represent the sample data points that we want to plot.
- ***plt.plot(x, y)***: This line creates a basic line plot using the plot function. It takes x and y as arguments to plot the data points.
- ***plt.xlabel('X-axis')*** and ***plt.ylabel('Y-axis')***: These lines label the X and Y axes, respectively, providing context for the plot. We usually give these more descriptive titles, for example the x-axis could be "time" and the y-axis could be "distance". Also good to add units after a slash so for example "Time / s" and "Distance / m".
- ***plt.title('Line Plot Example')***: This line adds a title to the plot.
- ***plt.show()***: This function displays the plot on the screen.

Exercise 1.1

You are given the monthly average temperatures (in degrees Celsius) in a city for a year. Create a line plot to visualize the temperature trend over the year. The data is provided in the form of two lists: months (containing the month names) and temperatures (containing the corresponding average temperatures).

months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
temperatures = [10, 11, 15, 18, 22, 25, 28, 28, 25, 20, 15, 11]

- Create a line plot that shows the temperature trend over the year.
- Add appropriate labels to the X and Y axes.
- Add a title to the plot to indicate what it represents.

1.2. Scatter Plot

Scatter plots are used to visualize individual data points as dots on a two-dimensional plane. They are valuable for identifying patterns, trends, and relationships between two variables.

We will create a scatter plot using Matplotlib to visualize individual data points.

Code:

```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 12, 5, 8, 7]

# Create a scatter plot
plt.scatter(x, y, color='red', marker='o', label='Data
Points')

# Add labels and a legend
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Scatter Plot Example')
plt.legend()

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

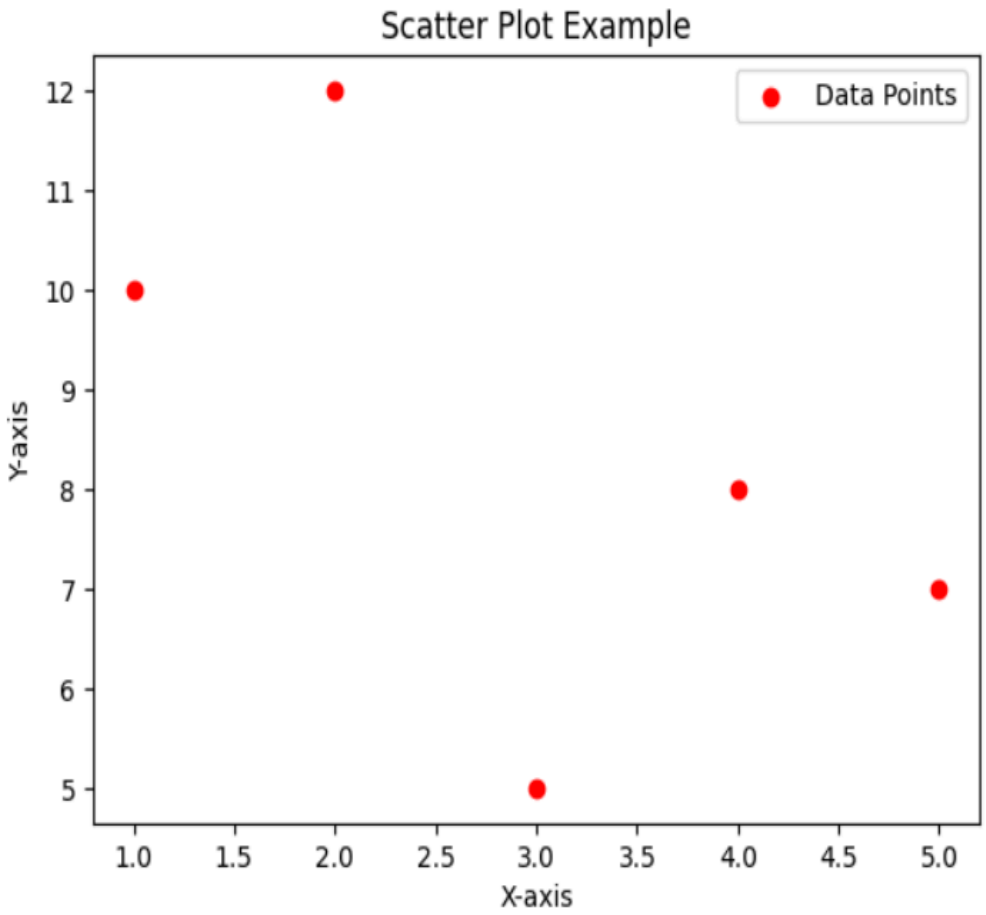

Figure 1: Scatter Plot

By examining the scatter plot, we observe that there isn't a definitive trend or pattern. The data points appear scattered across the plot, without a consistent upward or downward slope. The point at approximately (3, 5) stands out as an outlier, as it significantly deviates from the general grouping of points. Outliers can be important indicators of unique or unusual data.

Scatter plots excel at providing precise information about individual data points. They allow us to see the exact coordinates of each point, which can be critical in certain analyses. In this case, we're exploring the relationship between ***x*** and ***y*** values. However, without a clear trend, it's more challenging to draw conclusions about how changes in ***x*** affect ***y***.

Code Explanation

- We ***import Matplotlib as plt*** as before.
- ***x*** and ***y*** represent the sample data points.
- ***plt.scatter(x, y, color='red', marker='o', label='Data Points')***: This line creates a scatter plot. We specify the color, marker style (in this case, a red circle), and label for the data points.
- ***plt.xlabel('X-axis')*** and ***plt.ylabel('Y-axis')***: These lines label the X and Y axes, providing context for the plot.
- ***plt.legend()***: This line adds a legend to the plot, using the label provided in the scatter function.
- ***plt.show()***: This function displays the plot.

1.3. Bar Chart

Bar charts represent categorical data with discrete bars, making it easy to compare values across different categories. We use bar charts when we want to compare data across categories, show rankings, or display frequencies or counts for discrete items.

We will create a bar chart using Matplotlib to visualize categorical data and compare values.

Code:

```
import matplotlib.pyplot as plt

# Sample data
categories = ['Category A', 'Category B', 'Category C']
values = [25, 40, 30]

# Create a bar chart
plt.bar(categories, values, color='skyblue')
```

```
# Add labels and a title
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Bar Chart Example')

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

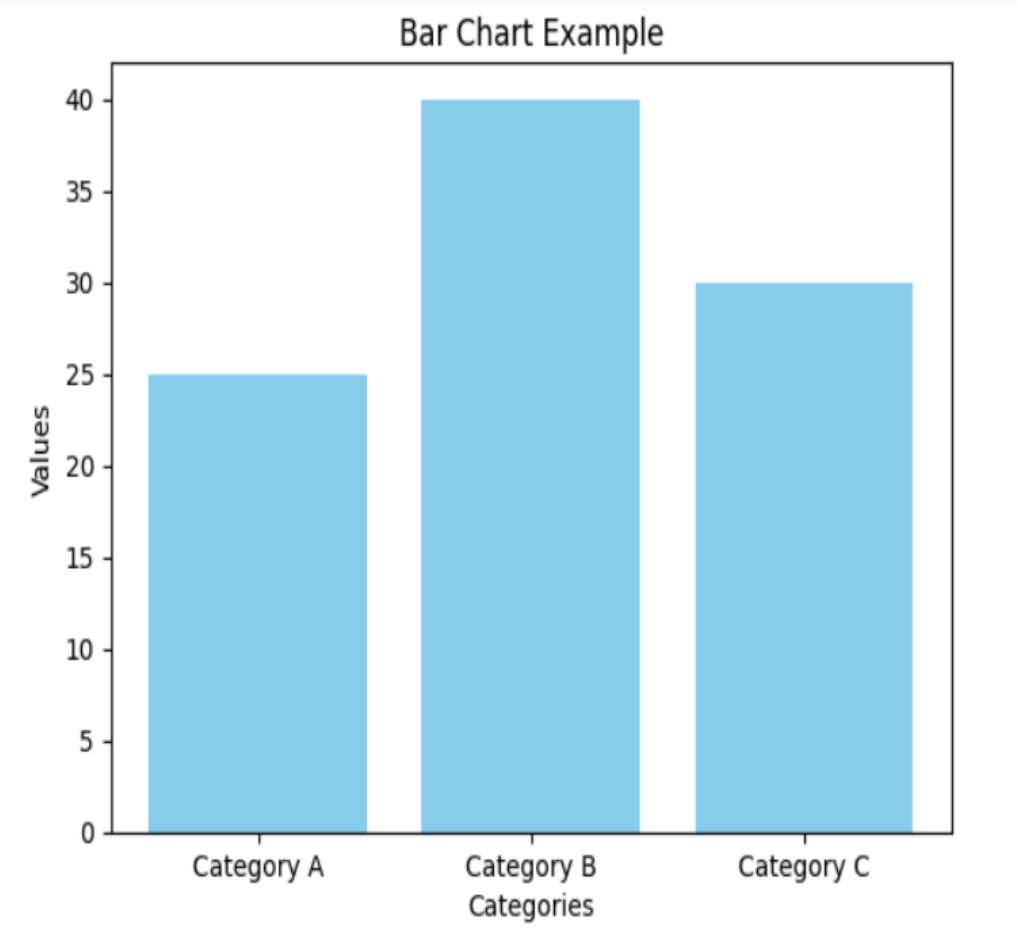

Figure 3: Bar Chart

The bar chart allows for a straightforward comparison of values between different categories. We can easily see that ‘Category B’ has the highest value (40), followed by ‘Category C’ (30) and ‘Category A’ (25). Each bar

represents a distinct category, making it easy to visually assess the relative magnitudes of each category. The vertical height of each bar directly corresponds to the value it represents. This provides a clear and intuitive representation of the data.

Code Explanation

- We ***import matplotlib.pyplot as plt*** as before.
- **categories** and **values** represent the categorical data and their corresponding values.
- ***plt.bar(categories, values, color='skyblue')***: This line creates a bar chart. We specify the categories, values, and color for the bars.
- ***plt.xlabel('Categories')*** and ***plt.ylabel('Values')***: These lines label the X and Y axes, providing context for the plot.
- ***plt.title('Bar Chart Example')***: This line adds a title to the plot.
- ***plt.show()***: This function displays the plot

1.4. Histogram

Histograms can represent both continuous and discrete values They divide the data into bins or intervals and show the frequency or count of data points in each bin. We use histograms when we want to understand statistics such as the shape of a dataset's distribution, identify central tendencies (mean, median, mode), and observe data skewness or the presence of multiple peaks.

We will create a histogram using Matplotlib to visualize the distribution of numerical data.

Code:

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

# Generate random data
data = np.random.randn(1000)

# Create a histogram
plt.hist(data, bins=20, color='green', alpha=0.6)

# Add labels and a title
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.title('Histogram Example')

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

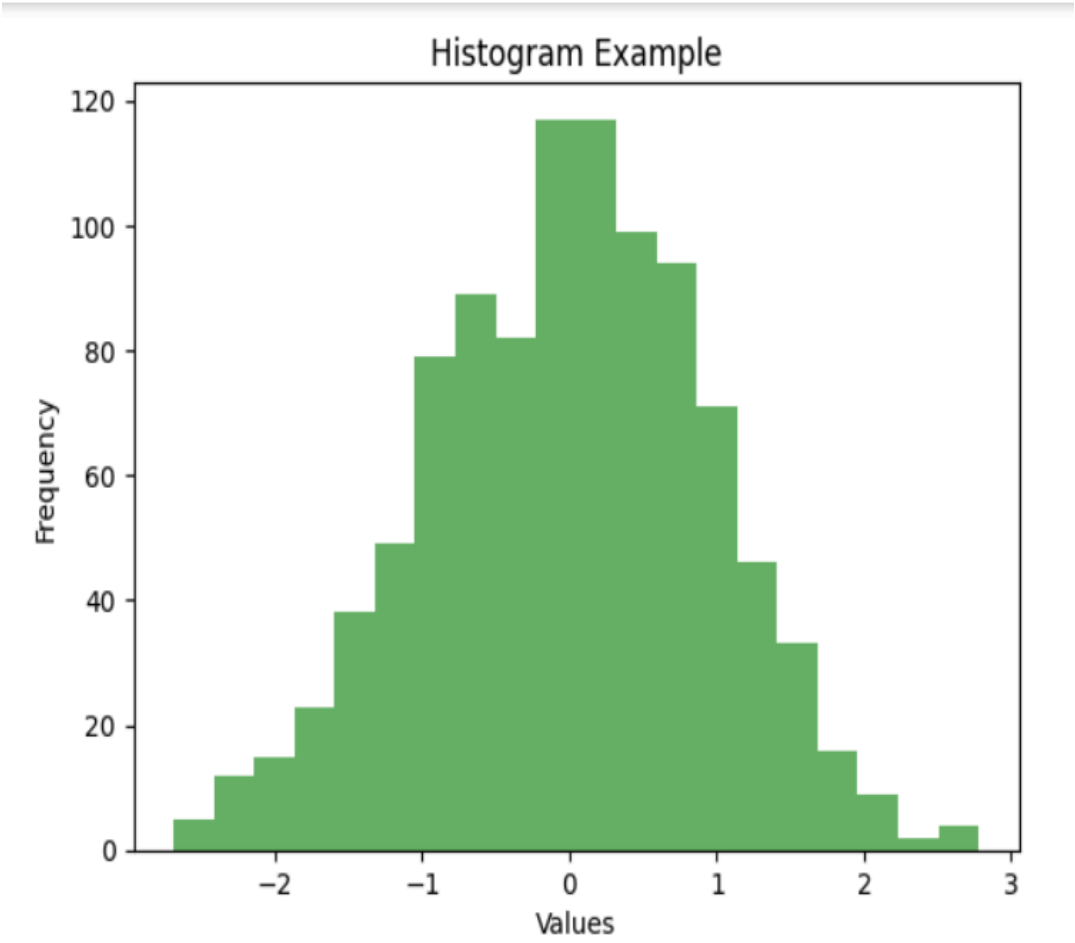

Figure 4: Histogram Example

The histogram provides insights into the shape of the data distribution. In this case, it appears to be approximately normally distributed, centered around zero. Each bin on the x-axis represents a range of values, and the height of the bars indicates how many data points fall within that range. For example, the tallest bar around zero indicates a higher frequency of values in that range. The spread of the bars shows the variability of the data. Wider distributions indicate more variability, while narrower distributions suggest less variation.

Also remember, usually high bars or isolated bars far from the main cluster may indicate outliers or anomalies in the data. The center of the

distribution (often represented by the highest point) gives an indication of the central tendency of the data.

Code Explanation

- We ***import matplotlib as plt*** as before and also ***import numpy as np*** for data generation.
- data is generated as random data using NumPy's ***randn*** function (np.random.randn samples from a normal distribution with mean zero and variance 1).
- ***plt.hist(data, bins=20, color='green', alpha=0.6)***: This line creates a histogram. We specify the data, the number of bins, the color of the bars, and their transparency (alpha).
- ***plt.xlabel('Values')*** and ***plt.ylabel('Frequency')***: These lines label the X and Y axes, providing context for the plot.
- ***plt.title('Histogram Example')***: This line adds a title to the plot.
- ***plt.show()***: This function displays the plot.

1.5. Pie Chart

Pie charts show the composition of a whole by dividing it into segments, typically representing proportions or percentages. We use pie charts when we want to visualize how individual components contribute to a whole, but be cautious with complex datasets or too many segments, as they can be challenging to interpret.

We will create a pie chart using Matplotlib to represent the distribution of categories within a dataset.

Code:

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

```
# Sample data
categories = ['Category A', 'Category B', 'Category C',
'Category D']
sizes = [30, 20, 15, 35]

# Create a pie chart
plt.pie(sizes, labels=categories, autopct='%1.1f%%',
startangle=90, colors=['gold', 'lightcoral',
'lightskyblue', 'lightgreen'])

# Add a title
plt.title('Pie Chart Example')

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

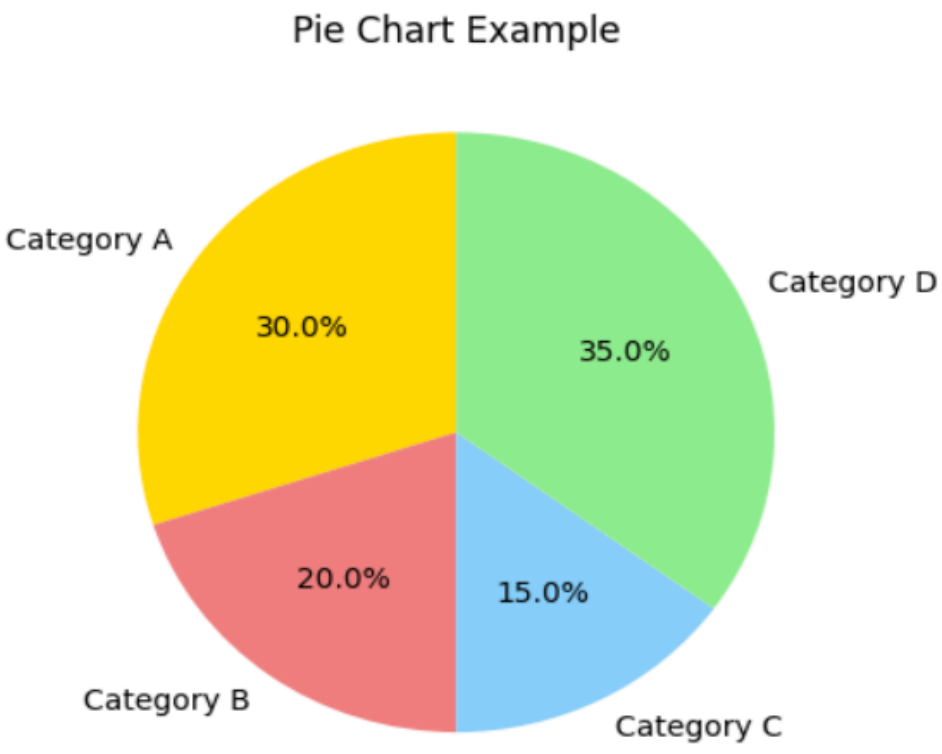

Figure 5: Pie Chart

The sizes of the wedges accurately represent the relative proportions of the categories. For example, 'Category D' (35%) has the largest segment, indicating it is the largest category.

The pie chart allows for an easy visual comparison of the different categories. Viewers can quickly see which category is the largest and how the others compare.

Code Explanation

- We ***import matplotlib as plt*** as before.
- **categories** and **sizes** represent the categories and their respective sizes.
- ***plt.pie(sizes, labels=categories, autopct='%1.1f%%', startangle=90, colors=['gold', 'lightcoral', 'lightskyblue', 'lightgreen'])***: This line creates a pie chart. We specify the sizes, labels, percentage format, starting angle, and colors for the pie slices.
- ***plt.title('Pie Chart Example')***: This line adds a title to the plot.
- ***plt.show()***: This function displays the pie chart.

Exercise 1.2

You have survey data that shows the distribution of favorite pizza toppings among a group of 100 people. Create a pie chart to visualize the distribution of pizza topping preferences. The data is provided in the form of a dictionary where each topping is a key, and the corresponding value is the number of people who chose that topping as their favorite.

```
pizza_toppings = {
   'Pepperoni': 30,
   'Mushroom': 20,
   'Margherita': 15,
   'Vegetarian': 10,
   'Supreme': 10,
   'Hawaiian': 5,
```

```
  'Other': 10
}
```

- Create a pie chart that displays the distribution of favorite pizza toppings.
- Label each slice of the pie chart with the topping name and the percentage of people who chose it as their favorite.
- Highlight the slice with the most popular topping using the explode parameter of Matplotlib.
- Add a title to the pie chart to describe its purpose.

1.6. Subplots

Subplots in data visualization refer to the division of a single figure into numerous smaller plots, allowing the display of various visualizations or related data representations inside a shared space at the same time. They allow for side-by-side or grid-based chart configurations, aiding comparisons, correlations, or juxtaposition of disparate datasets. Subplots are extremely useful for highlighting distinct parts of data or different data dimensions, boosting the viewer's understanding by offering several views within a consistent visual framework. It allows analysts and researchers to easily express complicated relationships, trends, or comparisons, improving the clarity and depth of data findings.

Let's walk through creating a figure with multiple subplots, each showcasing a different type of plot explained above.

Code:

```
import matplotlib.pyplot as plt

# Creating data for plots
x = [1, 2, 3, 4, 5]  # Simple x-axis values
y = [10, 7, 15, 9, 12]  # Corresponding y-axis values

# Creating figure and subplots
```

```
fig, axs = plt.subplots(2, 3, figsize=(12, 8))

# Line plot
axs[0, 0].plot(x, y)
axs[0, 0].set_title('Line Plot')

# Scatter plot
scatter_y = [8, 6, 12, 7, 10]  # Simple y-axis values for
scatter plot
axs[0, 1].scatter(x, scatter_y)
axs[0, 1].set_title('Scatter Plot')

# Bar chart
categories = ['A', 'B', 'C', 'D']
values = [7, 3, 9, 5]
axs[0, 2].bar(categories, values)
axs[0, 2].set_title('Bar Chart')


# Histogram
hist_data = [5, 7, 6, 8, 5, 7, 8, 6, 7, 8, 6, 5, 7, 6, 8,
7, 5, 7, 6, 8, 7, 5, 6, 8]  # Simple histogram data
axs[1, 0].hist(hist_data, bins=5)
axs[1, 0].set_title('Histogram')

# Pie chart
sizes = [30, 20, 25, 15, 10]
labels = ['Apple', 'Orange', 'Banana', 'Grape', 'Melon']
axs[1, 1].pie(sizes, labels=labels, autopct='%1.1f%%')
axs[1, 1].set_title('Pie Chart')

# Removing empty subplot
fig.delaxes(axs[1, 2])

# Adjusting layout
plt.tight_layout()

# Showing the plots
```

```
plt.show()
```

Code Explanation

- **fig, axs = plt.subplots(2, 3, figsize=(12, 8))** generates a figure with a 2x3 grid of subplots and sets its size to 12x8 inches.
- **Line Plot:**
  - **axs[0, 0].plot(x, y):** Plots a line graph using x as the x-axis values and y as the y-axis values.
  - **axs[0, 0].set_title('Line Plot'):** Sets the title for the line plot subplot.
- **Scatter Plot:**
  - **scatter_y = [8, 6, 12, 7, 10]:** Defines a list of y-axis values for a scatter plot.
  - **axs[0, 1].scatter(x, scatter_y):** Creates a scatter plot using x values against scatter_y.
- **Bar Chart:**
  - **categories = ['A', 'B', 'C', 'D']:** Represents categories for the bar chart.
  - **values = [7, 3, 9, 5]:** Defines values corresponding to each category.
  - **axs[0, 2].bar(categories, values):** Generates a bar chart with categories on the x-axis and values on the y-axis.
- **Histogram:**
  - **hist_data:** Represents data for the histogram.
  - **axs[1, 0].hist(hist_data, bins=5):** Creates a histogram with hist_data and 5 bins.
- **Pie Chart:**
  - **sizes:** Contains sizes representing portions of a pie chart.
  - **labels:** Represents labels for each portion in the pie chart.
  - **axs[1, 1].pie(sizes, labels=labels, autopct='%1.1f%%'):** Generates a pie chart with specified sizes and labels, displaying percentages.
  - **axs[1, 1].set_title('Pie Chart'):** Sets the title for the pie chart subplot.
- **fig.delaxes(axs[1, 2])** removes an empty subplot at position (1, 2).

- **plt.tight_layout()** adjusts the layout of subplots to avoid overlapping.
- **plt.show()** displays the generated subplots containing the line plot, scatter plot, bar chart, histogram, and pie chart.

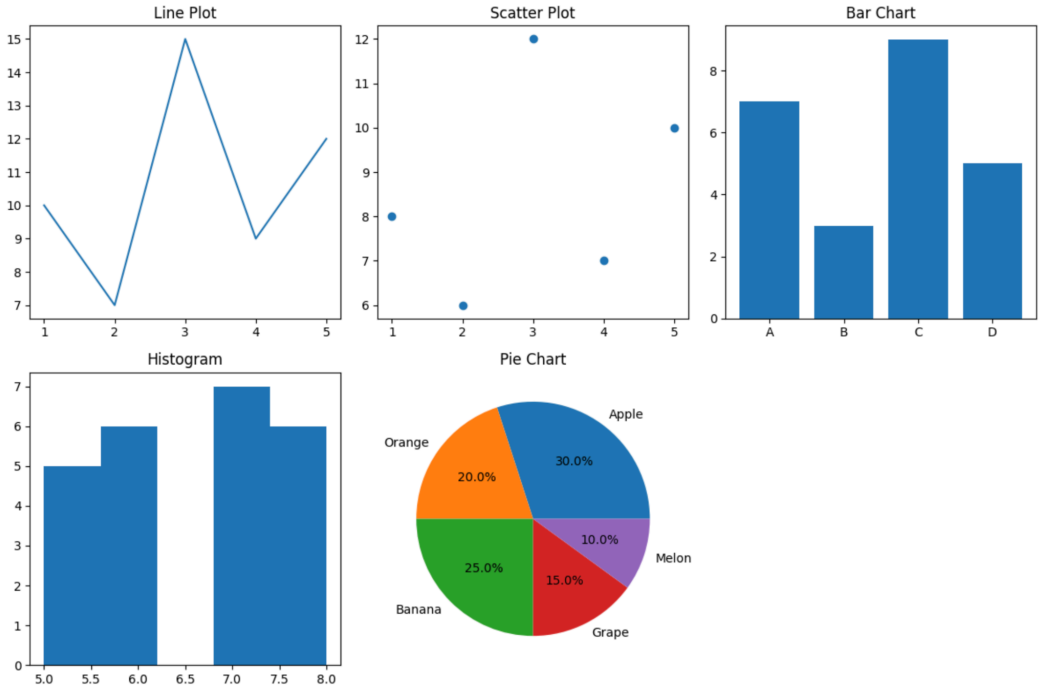

Figure 6: Subplots

2. Seaborn

Seaborn is a popular Python data visualization library that is often used in conjunction with Matplotlib. While Matplotlib is a powerful and flexible library for creating a wide range of basic and advanced plots, Seaborn is designed specifically for statistical data visualization. Seaborn builds on top of Matplotlib and provides several advantages:

- High-Level Interface: Seaborn offers a higher-level, more concise interface for creating complex statistical visualizations with minimal code. It simplifies many common visualization tasks.

- Built-in Themes: Seaborn comes with several built-in themes and color palettes that can make your plots look more aesthetically pleasing with minimal effort.
- Statistical Plotting: Seaborn includes specialized functions for creating informative statistical plots such as violin plots, box plots, and heatmaps. These functions make it easier to visualize and understand the underlying data distribution.
- Integration with Pandas: Seaborn is designed to work seamlessly with Pandas DataFrames, which is the primary data structure for data manipulation in Python. This makes it convenient for working with real-world datasets.

Let's explore some statistical visualization examples using Seaborn, along with step-by-step explanations for each one.

2.1. Box Plot

Box plots provide a summary of the distribution and spread of data, helping identify outliers and understand the data's central tendency. We use box plots when we want to compare the distribution of data across different categories or groups, especially when dealing with skewed or non-normal data.

The box in the plot represents the interquartile range (IQR), which contains the middle 50% of the data. The lower edge of the box represents the 25th percentile (Q1), and the upper edge represents the 75th percentile (Q3). Inside the box, a horizontal line represents the median (50th percentile) of the data. Lines extending from the box indicate the range of non-outlier data. They typically extend to the minimum and maximum values within 1.5 times the IQR. Individual data points outside the whiskers are considered outliers and are plotted individually.

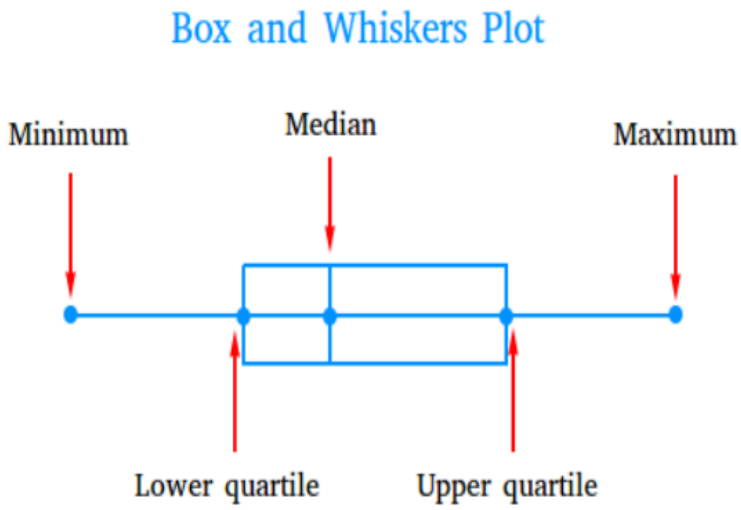

Figure 6: Box Plot Explanation (Image Source)

Now, we will create a box plot using Seaborn to visualize the distribution and spread of a dataset.

Code:

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

# Sample data
data = sns.load_dataset("tips")

# Create a box plot
sns.boxplot(x="day", y="total_bill", data=data,
palette="husl")

# Add a title
plt.title('Seaborn Box Plot Example')

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

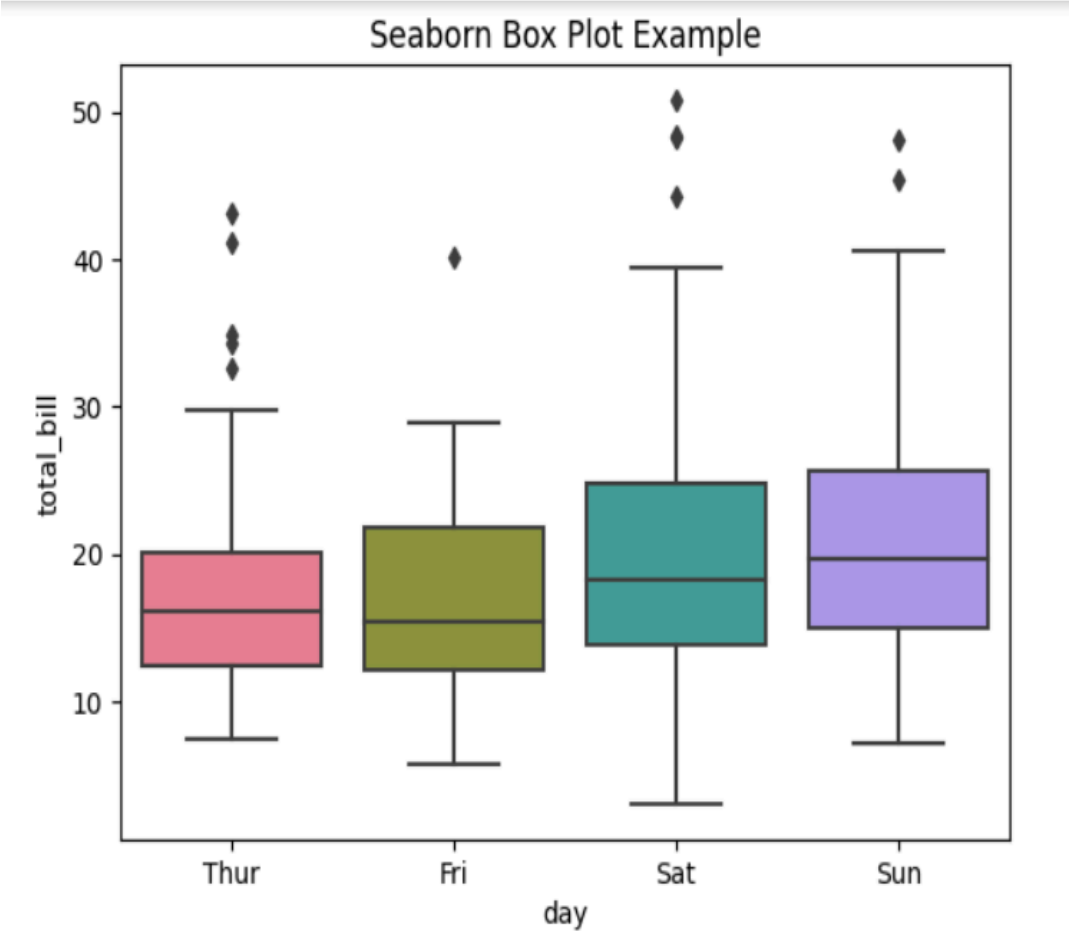

Figure 7: Box Plot

The box plot provides a clear representation of the distribution of total bills for each day. It allows us to compare the central tendency, spread, and potential outliers across different days.

The position of the median line within the box indicates the central tendency of the data for each day. For example, 'Sun' appears to have the highest median total bill.

The size of the box and the length of the whiskers indicate the variability of the data. A larger box and longer whiskers suggest greater variability.

Individual data points outside the whiskers are potential outliers. For example, 'Thur' has some high outliers, indicating unusually large total bills on that day.

Code Explanation

- We ***import Seaborn as sns*** and ***Matplotlib as plt***.
- We load a sample dataset (the Tips dataset in this case).
- ***sns.boxplot(x="day", y="total_bill", data=data, palette="husl")***: This line creates a box plot, displaying the distribution of total_bill for each day. The palette argument specifies the color palette.
- ***plt.title('Seaborn Box Plot Example')***: This line adds a title to the plot.
- ***plt.show()***: This function displays the box plot.

2.2. Heatmap

Heatmaps are used to visualize relationships and patterns in a matrix by color-coding values. We use heatmaps to display correlation matrices, visualize hierarchies, or explore patterns in large datasets where it's important to see relationships between variables. Each cell in the heatmap is colored based on the value it represents. Darker colors indicate higher values, while lighter colors indicate lower values.

We will create a heatmap using Seaborn to visualize a correlation matrix.

Code:

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

# Sample data: correlation matrix
data = sns.load_dataset("flights")
```

```
corr_matrix = data.pivot_table(index="month",
columns="year", values="passengers")

# Create a heatmap
sns.heatmap(corr_matrix, cmap="coolwarm")

# Add a title
plt.title('Seaborn Heatmap Example')

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

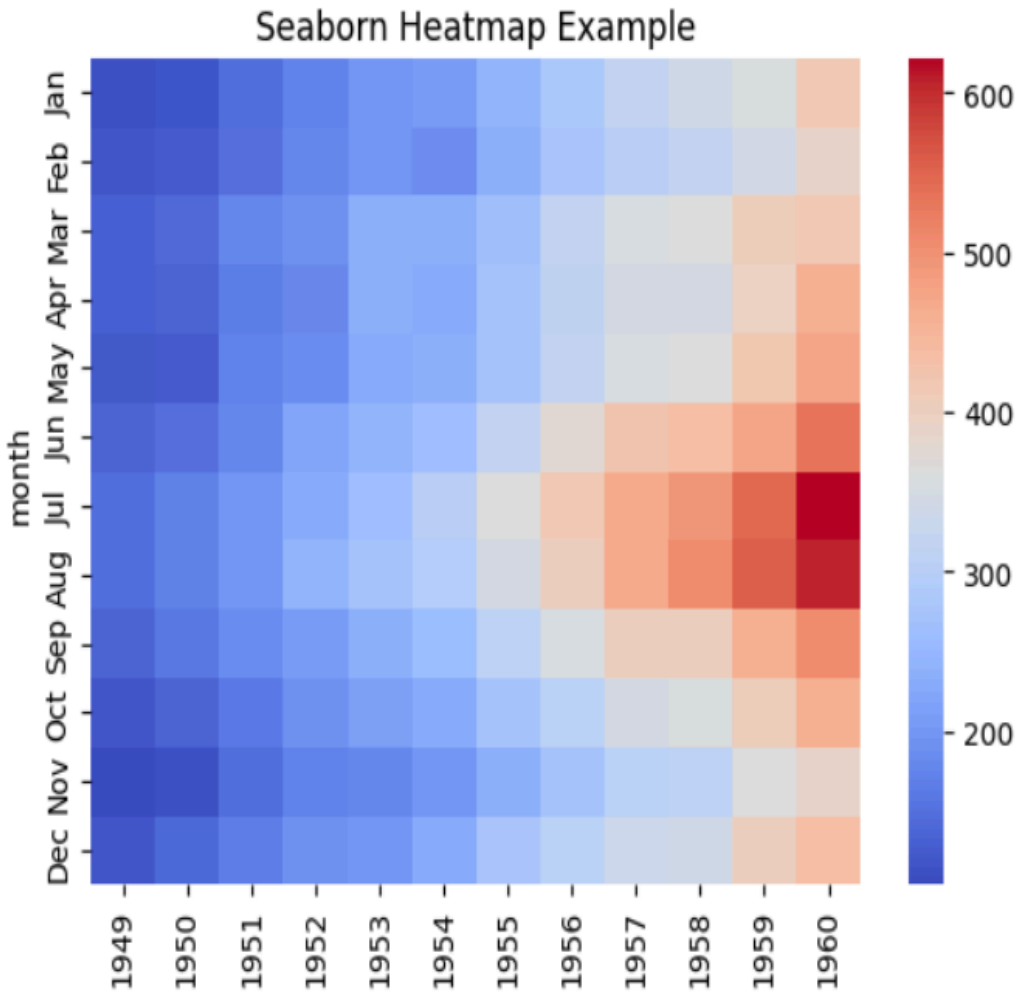

Figure 8: Heatmap Example

The heatmap allows us to observe seasonal trends in passenger numbers. For example, there is a clear increase in passengers during the summer months of June, July, and August.

By looking across the years, we can identify patterns and changes in passenger numbers. For instance, there seems to be a gradual increase in passengers over the years. Darker cells indicate higher passenger counts, while lighter cells indicate lower counts. This provides a visual representation of the variation in passenger numbers across months and years.

The darkest cell corresponds to July 1960, suggesting it had the highest number of passengers during the observed period. Conversely, the lightest cells correspond to January and February of 1949, indicating lower passenger numbers.

Code Explanation

- We ***import seaborn as sns*** and ***matplotlib as plt***.
- We load a sample dataset (the Flights dataset in this case) and create a correlation matrix using ***pivot_table()***.
- ***sns.heatmap(corr_matrix, annot=True, cmap="coolwarm")***: This line creates a heatmap that visualizes the correlation matrix. The annot argument adds numerical annotations to the cells, and the cmap argument specifies the color map.
- ***plt.title('Seaborn Heatmap Example')***: This line adds a title to the plot.
- ***plt.show()***: This function displays the heatmap.

2.3. Scatter Plot with Regression Line

In this example, Seaborn's regplot function is used to create a scatter plot with a regression line. The regression line provides a visual representation of the linear relationship between ***x*** and ***y***.

The regression line can be used to make predictions about ***y*** based on a given value of ***x***. However, the accuracy of these predictions depends on the strength of the linear relationship. The closeness of the data points to the regression line indicates the strength of the linear relationship. A tighter clustering around the line suggests a stronger correlation.

Seaborn automatically calculates and displays the linear regression line along with confidence intervals, providing a deeper level of analysis compared to a basic scatter plot created with Matplotlib alone.

Code:

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

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 12, 5, 8, 7]

# Create a scatter plot with a regression line
sns.regplot(x=x, y=y, color='blue', marker='o')

# Add labels and a title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Seaborn Scatter Plot Example')

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

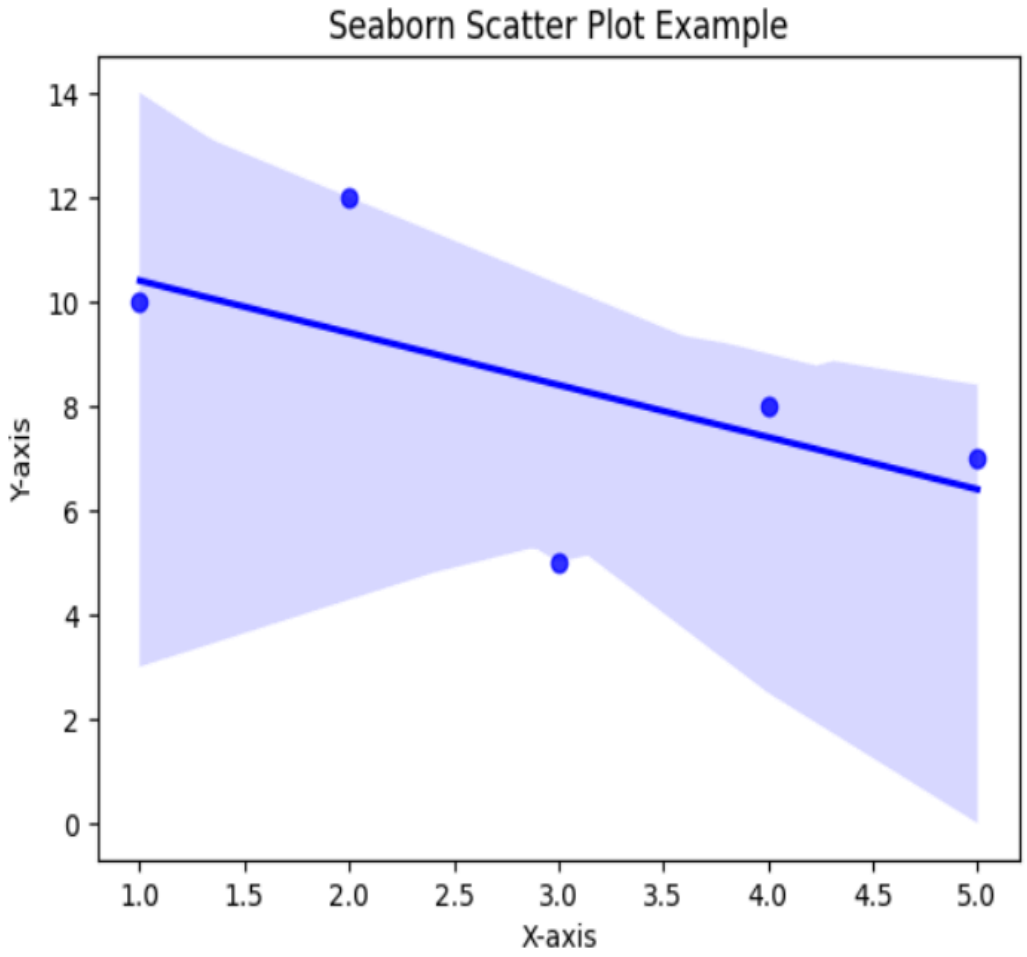

Figure 9: Scatter Plot with Regression Line

Exercise 2.1

- Describe the above scatter plot with the regression line.

3. Plot and Save

Saving a figure during visualization is crucial to preserve and share the insights gained from the data. It ensures that the visual representation is accessible for future reference, analysis, and communication with others.

Here's an example of how to create a simple line plot using Matplotlib and save it to an image file:

Code:

```
import matplotlib.pyplot as plt

# Sample data
x = [1, 2, 3, 4, 5]
y = [10, 12, 5, 8, 7]

# Create a line plot
plt.plot(x, y)

# Add labels and a title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Line Plot Example')

# Save the plot to an image file (e.g., PNG)
plt.savefig('line_plot.png')

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

Note

The **savefig** function in Matplotlib is used to save a Matplotlib figure (plot) as an image file in various formats, such as PNG, JPEG, PDF, SVG, and more. It allows you to customize the output file format, resolution, and other parameters. Here are some key points about the savefig function:

- # Save the plot as a PNG image with higher resolution and a white background
- plt.savefig('my_plot.png', format='png', dpi=300, bbox_inches='tight', facecolor='white')

Exercise 3.1

You have collected data on the distribution of favorite movie genres among a group of 200 people. The data is provided in the form of a dictionary where each genre is a key, and the corresponding value is the number of people who

prefer that genre. Create a data visualization to represent this distribution, and save it to a file for sharing.

```
movie_genres = {
   'Action': 45,
   'Comedy': 35,
   'Drama': 30,
   'Adventure': 25,
   'Sci-Fi': 20,
   'Romance': 15,
   'Horror': 10
}
```

- Create an appropriate data visualization using ‘seaborn’ to represent the distribution of favorite movie genres.
- Save the visualization to an image file (choose an appropriate file format) with a descriptive filename.
- Include labels, titles, or any other necessary annotations to make the visualization informative.

Note

- Most plotting libraries (e.g., Matplotlib and Seaborn) offer extensive customization options for adjusting colors, markers, labels, titles, and other plot elements.
- Different types of plots (e.g., histograms, box plots, scatter plots) are valuable for exploring data characteristics such as distribution, central tendency, outliers, and relationships.
- Choose the appropriate plot type based on whether your data is categorical (e.g., bar chart, pie chart) or continuous (e.g., line plot, histogram).
- Techniques like color mapping and subplotting can help visualize relationships in multivariate datasets.
- Seaborn builds on Matplotlib and provides simplified functions for creating attractive statistical plots.

- Some libraries (e.g., Plotly) offer interactive plots, allowing users to explore data dynamically.
- Select plots that effectively convey the message you want to communicate, considering your data's nature and your audience's needs.
- **Always label your axes, add legends, and include titles to make plots informative.**
- Familiarize yourself with popular Python plotting libraries like Matplotlib, Seaborn, Plotly, and others to choose the one that best suits your visualization needs.

4. Exploratory Data Analysis

We'll also dive into Exploratory Data Analysis (EDA), where we'll break down complex things into simpler parts. This includes looking at one thing at a time (univariate analysis) and comparing two things together (bivariate analysis). Through easy examples and hands-on tasks, we'll grasp how to uncover hidden patterns, spot connections between factors, and draw meaningful insights from data. This will empower us to make informed decisions and tell compelling stories using our data.

EDA serves as a foundational step in the data analysis process, aiding in understanding the characteristics and relationships within a dataset. The primary goal of EDA is to reveal insights, patterns, and anomalies that might not be immediately apparent, thereby guiding further analysis and decision-making.

4.1. Univariate Analysis: Understanding the Essence of Individual Variables

Imagine exploring a lush forest, each tree standing on its own, unique in shape and size. Univariate analysis is akin to examining each tree with a curious eye, allowing us to appreciate its distinct characteristics. In the realm of data analysis, univariate analysis involves focusing our attention on a single variable at a time. Just as each tree holds its story within its

branches, leaves, and bark, each variable in a dataset has its own tale to tell.

During univariate analysis, we study the behavior of one variable in isolation. This reveals its distribution, its typical value (central tendency), and the extent of its variation (dispersion). This process is like taking a magnifying glass to a single leaf and studying its color, texture, and shape. We can also spot potential outliers—values that stand apart from the rest, much like an unusual blossom catching our attention in the forest.

Univariate analysis also uncovers missing values—those gaps in our data—as if noticing gaps in the canopy where sunlight filters through. This awareness prompts us to address these gaps, ensuring our data is complete and reliable.

By understanding each variable's essence, we gain a holistic view of our data. This knowledge is akin to mapping the unique features of the forest, allowing us to navigate its depths with confidence. Armed with insights from univariate analysis, we embark on a journey to create clearer data, unravel intriguing patterns, and set the stage for more complex analyses.

Let's consider an example of univariate analysis using a simple dataset of exam scores. We'll use Python, Pandas, and Matplotlib libraries to perform the analysis. We'll calculate the basic statistics such as mean, median, standard deviation, minimum, and maximum scores, create a histogram to visualize the score distribution and compute the count and percentages for different score ranges for the exam scores.

Code:

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

# Create a sample dataset of exam scores
```

```
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Emma'],
        'Exam_Score': [85, 72, 90, 78, 92]}
df = pd.DataFrame(data)

# Calculate basic statistics for exam scores
mean_score = df['Exam_Score'].mean()
median_score = df['Exam_Score'].median()
std_deviation = df['Exam_Score'].std()
min_score = df['Exam_Score'].min()
max_score = df['Exam_Score'].max()

# Create a histogram to visualize score distribution
plt.hist(df['Exam_Score'], bins=10, edgecolor='black')
plt.title('Exam Score Distribution')
plt.xlabel('Score')
plt.ylabel('Frequency')
plt.show()

# Save the histogram as an image
plt.savefig('histogram.png')

# Calculate count and percentages for different score ranges
score_ranges = [0, 60, 70, 80, 90, 100]
score_counts = pd.cut(df['Exam_Score'], score_ranges,
include_lowest=True).value_counts()
score_percentages = (score_counts / len(df)) * 100

# Display the results
print("Exam Scores:")
print(df)
print("\nBasic Statistics:")
print("Mean Score:", mean_score)
print("Median Score:", median_score)
print("Standard Deviation:", std_deviation)
print("Minimum Score:", min_score)
print("Maximum Score:", max_score)
print("\nScore Distribution:")
print(score_counts)
```

```
print("\nPercentage Distribution:")
print(score_percentages)
```

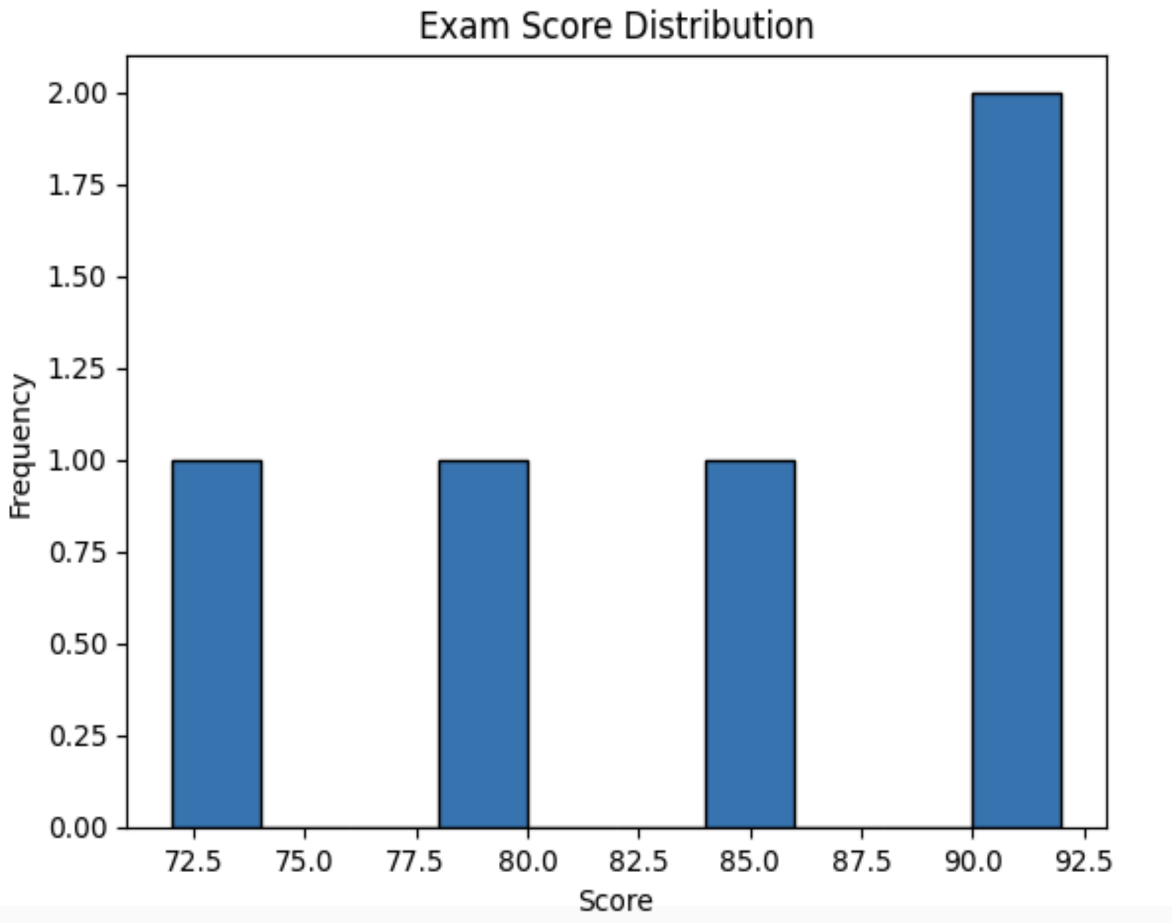

Figure 1: Exam Score Distribution

Output:

```
Exam Scores:
      Name  Exam_Score
0    Alice          85
1      Bob          72
2  Charlie          90
3    David          78
4     Emma          92

Basic Statistics:
Mean Score: 83.4
```

```
Median Score: 85.0
Standard Deviation: 8.35463942968217
Minimum Score: 72
Maximum Score: 92

Score Distribution:
Exam_Score
(70.0, 80.0]      2
(80.0, 90.0]      2
(90.0, 100.0]     1
(-0.001, 60.0]    0
(60.0, 70.0]      0
Name: count, dtype: int64

Percentage Distribution:
Exam_Score
(70.0, 80.0]      40.0
(80.0, 90.0]      40.0
(90.0, 100.0]     20.0
(-0.001, 60.0]     0.0
(60.0, 70.0]       0.0
Name: count, dtype: float64
```

Explanation

In the example, we visualized a histogram of the score distribution. A histogram is a graphical representation that displays the distribution of data. It groups data into intervals, or "bins", and shows how many data points fall into each bin. In the example, we're using the **plt.hist()** function from the Matplotlib library to create a histogram of the exam scores. Here's what the histogram does:

- Binning: The data range is divided into several bins. In this case, we used 10 bins to represent the score distribution.
- Frequency: The height of each bar in the histogram represents the frequency (count) of data points that fall within that bin's range. The taller the bar, the more data points it represents.

- Visualization: The histogram provides a visual representation of how scores are spread across different score ranges. It helps you understand the concentration of scores around certain values and the overall shape of the distribution.

Explanation

Also, in this example, we calculated the count and percentages of scores falling within specific ranges. Here's how this part works

- Score Ranges: We defined score ranges, such as (80.0, 90.0], (70.0, 80.0], etc. These ranges are intervals that group scores into different categories.
- pd.cut() and value_counts(): We used the Pandas function pd.cut() to categorize exam scores into the defined ranges. Then, value_counts() was used to count the number of scores in each range.
- Percentage Calculation: To understand the distribution relative to the entire dataset, we calculated the percentages of scores in each range by dividing the count for each range by the total number of scores and multiplying by 100.

This count and percentage distribution helps you grasp the proportion of scores falling within different score ranges. It allows you to see how much of the data is concentrated in specific regions and how spread out or compact the distribution is.

Exercise 3.1

- Please go to the main program you downloaded, and solve question number 6.

4.2. Bivariate Analysis: Discovering Relationships Between Variables

Picture a dance where two partners move in harmony, their steps influencing each other's rhythm. Bivariate analysis is like observing this dance between variables in a dataset, where one variable's movement

can gracefully guide another's. Another example can be study hours and sleep hours effect on the final exam scores.

Bivariate analysis involves a dynamic exploration of two variables together, seeking to uncover connections that might go unnoticed when observing them individually. Just as in a duet, where partners respond to each other's cues, bivariate analysis helps us discern how changes in one variable relate to shifts in another.

Imagine we're comparing the heights and weights of individuals. The bivariate analysis would help us discover whether taller people tend to be heavier or if there's no clear pattern. This exploration is like watching the dancers' movements intertwine, trying to understand the nature of their interaction.

Through bivariate analysis, we identify correlations, which are akin to recognizing how closely the dancers follow each other's steps. A strong correlation suggests a tight bond, while a weak one indicates a more casual connection. This awareness helps us predict how one variable might change if we know the other's behavior.

Moreover, bivariate analysis allows us to uncover potential cause-and-effect relationships. It's like observing how a change in the music tempo affects the dancers' movements. By studying these interactions, we can make informed assumptions about how changes in one variable influence another's behavior.

Bivariate analysis is a powerful tool for unveiling the hidden dynamics within our data. Just as observing a dance can reveal the nuances of a partnership, bivariate analysis helps us see how variables interact, guiding us to deeper insights, informed decision-making, and richer storytelling through data.

Let's explore a bivariate analysis example using Python, Pandas, Matplotlib, and Seaborn libraries. In this example, we'll use a dataset containing the heights and weights of individuals to investigate the relationship between these two variables.

Code:

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

# Create a sample dataset of heights and weights
data = {'Height (inches)': [65, 68, 70, 61, 72, 63, 67, 66, 69,
71],
        'Weight (lbs)': [150, 160, 180, 130, 200, 140, 155, 148,
170, 190]}
df = pd.DataFrame(data)

# Calculate the correlation between height and weight
correlation = df['Height (inches)'].corr(df['Weight (lbs)'])

# Create a scatter plot to visualize the relationship
sns.scatterplot(data=df, x='Height (inches)', y='Weight (lbs)')
plt.title('Height vs Weight')
plt.xlabel('Height (inches)')
plt.ylabel('Weight (lbs)')
plt.show()

# Display the correlation coefficient
print("Correlation between Height and Weight:", correlation)
```

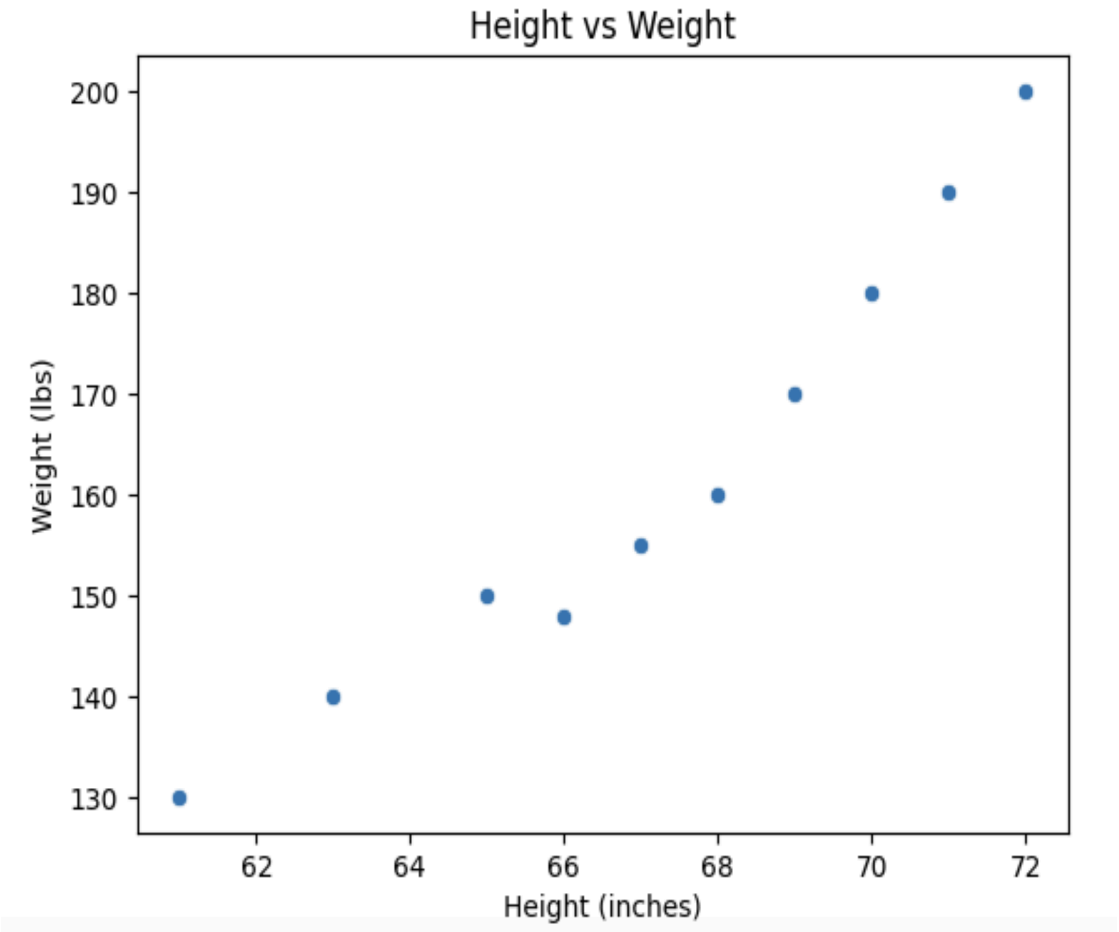

Figure 2: Height vs. Weight

Output:

```
Correlation between Height and Weight: 0.9682681701295192
```

Exercise 3.2

- Please go to the main program you downloaded, and solve question number 7.

Note

During this session, we've covered some data preprocessing and EDA techniques. In future data visualization, and ML sessions, we'll learn more. However, remember, the data world is vast. Stay curious, stay adaptive, and

keep honing your data preprocessing and EDA skills to become a proficient data analyst. Happy learning!

Key Points

- **plt.xlabel()** and **plt.ylabel()** are used to label the X and Y axes, providing context for the plot.
- **plt.title()** adds a title to the plot, summarizing its purpose or content.
- Scatter plots (**plt.scatter()**) are valuable for visualizing individual data points and identifying patterns or relationships.
- Histograms (**plt.hist()**) help visualize data distributions and identify central tendencies.
- Box plots (**plt.boxplot()**) provide a summary of data spread, identifying outliers and quartiles.
- Heatmaps (**sns.heatmap()**) are used to display correlation matrices and visualize patterns in large datasets.
- Bar charts (**plt.bar()**) are suitable for comparing data across categories or groups.
- Pie charts (**plt.pie()**) illustrate proportions within a whole, useful for categorical data.

Further Resources

Pandas by Official Documentation
Seaborn by Official Documentation
Matplotlib by Official Documentation
Matplotlib Tutorial by w3schools
Visualization with Seaborn by Jake VanderPlas