1 a)Write a python program to find the best of two test average marks out of three test's marks accepted from the user.

Constraints:

- The program assumes that the user will enter valid numerical test scores.
- The program calculates the average using the provided test scores.
- The best of two averages is calculated by considering two out of the three test scores.

Input:

- Enter the score for Test 1: 85
- Enter the score for Test 2: 92
- Enter the score for Test 3: 78

Output:

- Average of all three tests: 85.00
- Best of two test averages: 90.00

Program:

```
def calculate_average(test_scores):
   return sum(test_scores) / len(test_scores)
# Input: Accept three test scores from the user
test_scores = []
for i in range(3):
   score = float(input(f"Enter the score for Test {i + 1}: "))
   test_scores.append(score)
# Calculate the average of all three test scores
average_all = calculate_average(test_scores)
# Calculate the best of two test averages
best_of_two = max(calculate_average(test_scores[:2]),
calculate_average(test_scores[1:]))
```

```
# Output the results
print(f"Average of all three tests: {average_all:.2f}")
print(f"Best of two test averages: {best_of_two:.2f}")
```

Output:

```
Enter the score for Test 1: 45
Enter the score for Test 2: 56
Enter the score for Test 3: 78
Average of all three tests: 59.67
Best of two test averages: 67.00
```

1b)Develop a Python program to check whether a given number is palindrome or not and also count the number of occurrences of each digit in the input number

Constraints:

- The program assumes that the input is a valid integer.
- The program works for both positive and negative numbers.
- The program is case-sensitive, so it distinguishes between different cases of letters and non-digit characters.

Input:

Enter a number: 12321

Output:

- 12321 is a palindrome.
- Digit 1 occurs 2 time(s) in the number.
- Digit 2 occurs 2 time(s) in the number.
- Digit 3 occurs 2 time(s) in the number.

Input:

- Enter a number: 12345

Output:

- 12345 is not a palindrome.
- Digit 1 occurs 1 time(s) in the number.
- Digit 2 occurs 1 time(s) in the number.
- Digit 3 occurs 1 time(s) in the number.
- Digit 4 occurs 1 time(s) in the number.
- Digit 5 occurs 1 time(s) in the number.

Program:

```
def is_palindrome(number):
    num_str = str(number)
    return num_str == num_str[::-1]
def count_digit_occurrences(number):
    digit_counts = {}
    num_str = str(number)
    for digit in num_str:
        if digit.isdigit():
            digit = int(digit)
            digit_counts[digit] = digit_counts.get(digit, 0) + 1
    return digit_counts
number = int(input("Enter a number: "))
if is_palindrome(number):
    print(f"{number} is a palindrome.")
else:
    print(f"{number} is not a palindrome.")
digit_counts = count_digit_occurrences(number)
for digit, count in digit_counts.items():
    print(f"Digit {digit} occurs {count} time(s) in the number.")
```

Output:

```
Enter a number: 123321
123321 is a palindrome.
Digit 1 occurs 2 time(s) in the number.
Digit 2 occurs 2 time(s) in the number.
Digit 3 occurs 2 time(s) in the number.
```

2 a) Defined as a function F as Fn = Fn-1 + Fn-2. Write a Python program which accepts a value for N (where N >0) as input and pass this value to the function. Display suitable error message if the condition for input value is not followed.

Constraints:

- The program checks whether N is a positive integer and provides an error message for invalid input.

Input:

- Enter a positive integer N: 7

Output:

- The 7-th Fibonacci number is: 8

Input:

- Enter a positive integer N: -5

Output:

- Invalid input. N must be greater than 0.

Input:

- Enter a positive integer N: abc

Output:

- Invalid input. Please enter a positive integer.

Program

```
def fibonacci(n):
    if n <= 0:
        return "Invalid input. N must be greater than 0."
    elif n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# Input: Accept a positive integer N from the user
try:
    N = int(input("Enter a positive integer N: "))
    if N <= 0:
        print("Invalid input. N must be greater than 0.")
    else:
        result = fibonacci(N)
        print(f"The {N}-th Fibonacci number is: {result}")
except ValueError:
    print("Invalid input. Please enter a positive integer.")
```

Output:

```
Enter a positive integer N: 10
The 10-th Fibonacci number is: 34
```

2b)Develop a python program to convert binary to decimal, octal to hexadecimal using functions.

Constraints:

- The program handles invalid input and provides error messages for incorrect binary or octal numbers.

Input:

- Enter a binary number: 1101
- Enter an octal number: 75

Output:

- Decimal value of 1101 is: 13
- Hexadecimal value of 75 is: 0xd

Input:

- Enter a binary number: 1234
- Enter an octal number: 123

Output:

- Invalid binary input.
- Hexadecimal value of 123 is: 0x7b

Program:

```
def binary_to_decimal(binary_str):
    try:
        decimal_value = int(binary_str, 2)
        return decimal_value
    except ValueError:
        return "Invalid binary input."
# Function to convert octal to hexadecimal
def octal_to_hexadecimal(octal_str):
    try:
        decimal_value = int(octal_str, 8)
        hexadecimal_value = hex(decimal_value)
        return hexadecimal_value
    except ValueError:
        return "Invalid octal input."
# Input: Accept user input for binary and octal numbers
binary_input = input("Enter a binary number: ")
octal_input = input("Enter an octal number: ")
# Convert binary to decimal
decimal_result = binary_to_decimal(binary_input)
print(f"Decimal value of {binary_input} is: {decimal_result}")
# Convert octal to hexadecimal
hexadecimal_result = octal_to_hexadecimal(octal_input)
print(f"Hexadecimal value of {octal_input} is: {hexadecimal_result}")
```

Output:

```
Enter a binary number: 0101
Enter an octal number: 567
Decimal value of 0101 is: 5
Hexadecimal value of 567 is: 0x177
```

3a)Write a Python program that accepts a sentence and find the number of words, digits, uppercase letters and lowercase letters.

```python
def analyze_sentence(sentence):

    # Initialize counters

    word_count = len(sentence.split())

    digit_count = 0

    uppercase_count = 0

    lowercase_count = 0

    # Iterate through each character in the sentence

    for char in sentence:

        if char.isalpha():

            if char.islower():

                lowercase_count += 1

            elif char.isupper():

                uppercase_count += 1

        elif char.isdigit():

            digit_count += 1

    # Display the results

    print("Number of words:", word_count)

    print("Number of digits:", digit_count)

    print("Number of uppercase letters:", uppercase_count)

    print("Number of lowercase letters:", lowercase_count)
```

```
# Input sentence from the user

user_sentence = input("Enter a sentence: ")

# Analyze the sentence

analyze_sentence(user_sentence)
```

Output:

```
Enter a sentence: Hello How are you doing? @123
Number of words: 6
Number of digits: 3
Number of uppercase letters: 2
Number of lowercase letters: 17
```

3b) Write a Python program to find the string similarity between two given strings

String similarity refers to the comparison or quantification of how alike two strings are. It is a measure of the degree of resemblance between two strings. There are various algorithms and metrics used to calculate string similarity, each with its own method of determining how similar or dissimilar two strings are.

Here are a few common methods for measuring string similarity:

- Levenshtein Distance (Edit Distance): This metric calculates the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into another. The similarity is often expressed as the inverse of the normalized Levenshtein distance.
- Jaccard Similarity: This measures the similarity between two sets by comparing the intersection and union of their elements. For strings, the sets are typically the sets of characters in each string.
- Cosine Similarity: Originally used for comparing documents, cosine similarity calculates the cosine of the angle between two vectors. In the context of strings, the strings are represented as vectors, and the similarity is computed based on the cosine of the angle between these vectors.
- Hamming Distance: This measures the minimum number of substitutions needed to change one string into the other, assuming the strings are of equal length. It's particularly useful when comparing strings of the same length.
- Jaro-Winkler Distance: Specifically designed for comparing strings that might have typos, this metric considers both the number of matching characters and the order of the characters, assigning higher weights to matching prefixes.

Program:

```
def get_cosine(vec1, vec2):
    # Find the common words (intersection) between the two vectors
    intersection = set(vec1.keys()) & set(vec2.keys())

    # Calculate the numerator (dot product) of the vectors
    numerator = sum([vec1[x] * vec2[x] for x in intersection])

    # Calculate the denominator (magnitude) of each vector
    sum1 = sum([vec1[x] ** 2 for x in list(vec1.keys())])
    sum2 = sum([vec2[x] ** 2 for x in list(vec2.keys())])

    # Calculate the square root of the product of the magnitudes
    denominator = math.sqrt(sum1) * math.sqrt(sum2)

    # Check if the denominator is zero to avoid division by zero
    if not denominator:
        return 0.0
    else:
        # Calculate and return the cosine similarity
        return float(numerator) / denominator
def text_to_vector(text):
    # Find all words in the text using the defined regular expression
    words = WORD.findall(text)

    # Use Counter to count the frequency of each word in the list
    return Counter(words)
s1 = "This is a foo bar sentence."
s2 = "This sentence is similar to a foo bar sentence."

# Convert the text strings into vectors
vector1 = text_to_vector(s1)
vector2 = text_to_vector(s2)

# Calculate the cosine similarity between the two vectors
cosine = get_cosine(vector1, vector2)

# Print the result
print(f"The cosine similarity between '{s1}' and '{s2}' is {cosine:.2f}.")
```

Output:

```
The cosine similarity between 'This is a foo bar sentence.' and 'This sentence is s
imilar to a foo bar sentence.' is 0.86.
```

4a) Write a Python program to Demonstrate how to Draw a Bar Plot using Matplotlib

Bar Plots:

A bar plot (or bar chart) is a graphical representation of data in which rectangular bars of varying lengths are used to represent different categories or groups. The length of each bar corresponds to the quantity or value it represents. Bar plots are effective for visualizing and comparing data across different categories, making them a popular choice for data analysis and presentation.

Key Components of Bar Plots

Bars:

The vertical or horizontal bars represent different categories or groups. The length (or height) of each bar is proportional to the value it represents.

Axis:

The axis provides a reference for the values associated with each bar. In a horizontal bar plot, the bars extend along the y-axis, and in a vertical bar plot, the bars extend along the x-axis.

Categories:

Each bar is associated with a specific category or group, and these categories are usually labeled on the axis.

Types of Bar Plots:

Vertical Bar Plot:

Bars are drawn vertically, extending along the x-axis. This type is suitable when comparing categories along a common baseline.

Horizontal Bar Plot:

Bars are drawn horizontally, extending along the y-axis. This type is effective when labels for categories are long or when comparing data across different groups.

Why Do We Need Bar Plots?

Comparison of Data:

Bar plots provide a straightforward way to compare the magnitudes of different categories. The lengths of the bars make it easy to visually assess the relative sizes of the data.

Categorical Data Representation:

Bar plots are ideal for representing categorical data, where each bar represents a distinct category or group. This makes them suitable for displaying counts, frequencies, or percentages associated with each category. Clarity and Simplicity:

Bar plots are simple to understand and visually intuitive. They communicate information in a clear and straightforward manner, making them accessible to a wide audience.

Effective Communication:

When presenting data to a non-technical audience or stakeholders, bar plots can be more effective than complex numerical tables or raw data. They provide a visual summary that is easy to interpret.

Trend Identification:

Trends and patterns in the data become apparent through the visual representation of bars. For example, it's easy to identify which category has the highest or lowest value.

Comparison Across Multiple Groups:

Bar plots can be used to compare data across multiple groups or subcategories. Grouped bar plots or stacked bar plots are common variations used for this purpose.

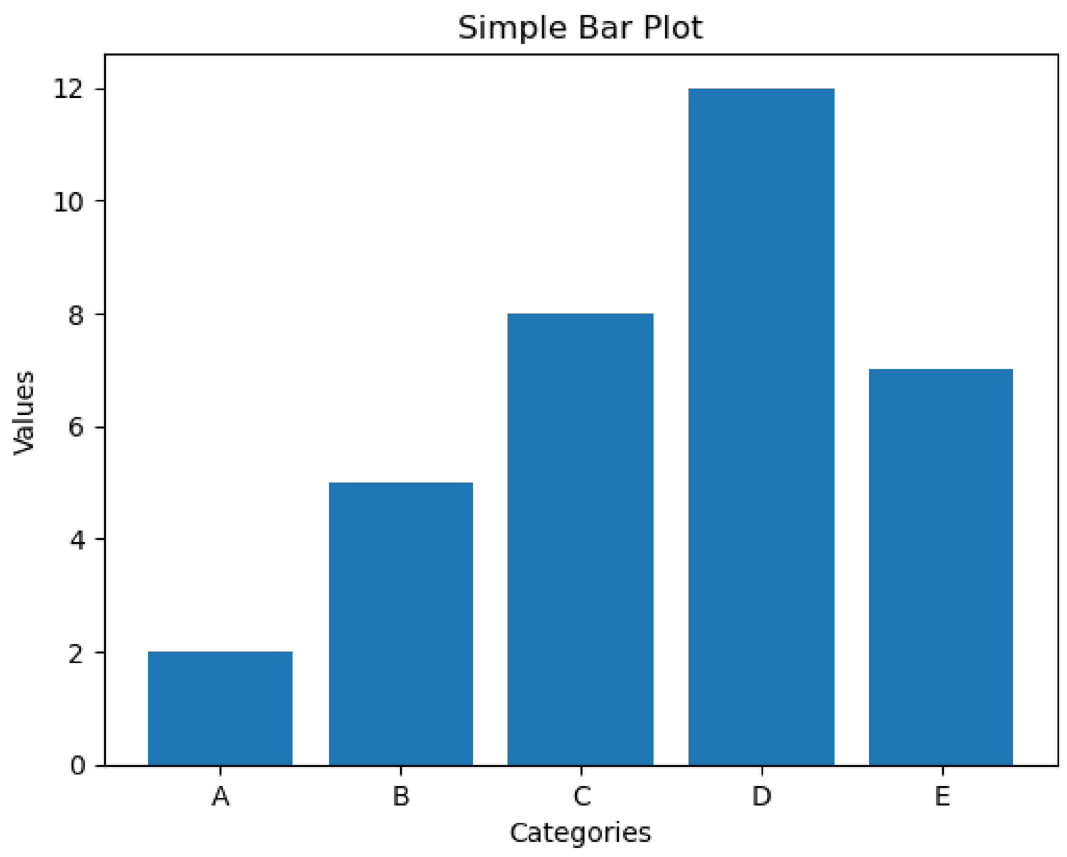
Frequency Distribution:

Bar plots are frequently used to display the frequency distribution of categorical data. This is particularly useful in fields such as statistics, market research, and social sciences. In summary, bar plots serve as effective tools for visualizing and communicating data in a way that is accessible and meaningful. Their simplicity, clarity, and ability to convey comparative information make them valuable in various fields, from scientific research to business analytics.

Program:

```
import matplotlib.pyplot as plt

data = [2, 5, 8, 12, 7]

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

plt.bar(categories, data)

plt.xlabel('Categories')

plt.ylabel('Values')

plt.title('Simple Bar Plot')

plt.show()
```

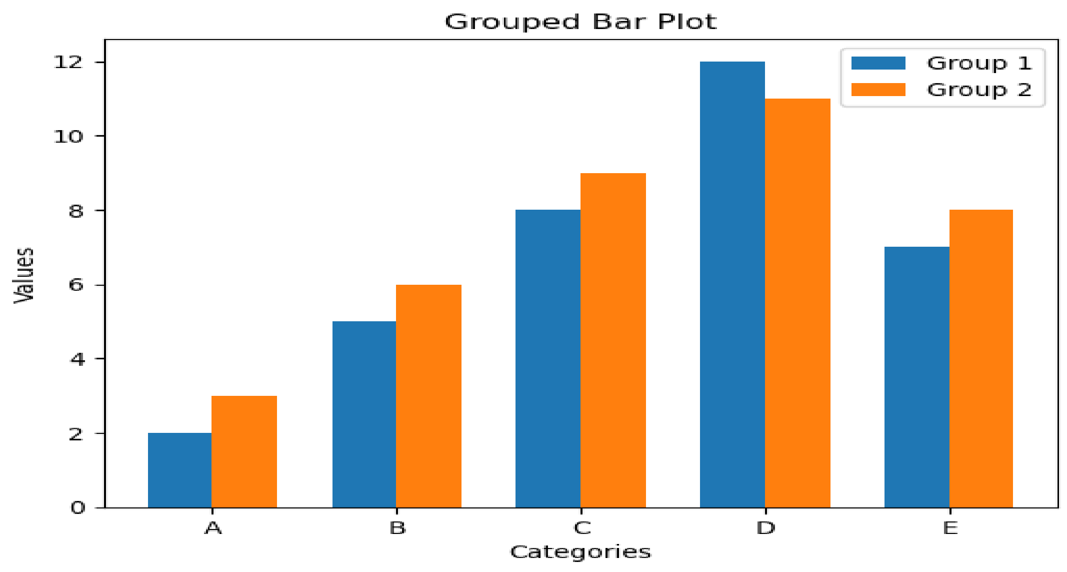
Output:



Group bar plot:

```
import matplotlib.pyplot as plt

import numpy as np

data1 = [2, 5, 8, 12, 7]

data2 = [3, 6, 9, 11, 8]

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

width = 0.35

fig, ax = plt.subplots()

bar1 = ax.bar(np.arange(len(categories)), data1, width, label='Group 1')

bar2 = ax.bar(np.arange(len(categories)) + width, data2, width, label='Group 2')

ax.set_xticks(np.arange(len(categories)) + width / 2)

ax.set_xticklabels(categories)

ax.legend()

plt.xlabel('Categories')

plt.ylabel('Values')

plt.title('Grouped Bar Plot')

plt.show()
```

Output:



4b)Write a python program to demonstrate how to draw a scatter plot using matplotlib

- What and why of scatter plot ?:

A scatter plot is a type of chart or graph used to visually represent the relationship between two numeric variables. It uses dots or other markers to represent individual data points on a two-dimensional plane, where the position of each point on the x and y axes corresponds to the values of the two variables for that data point.

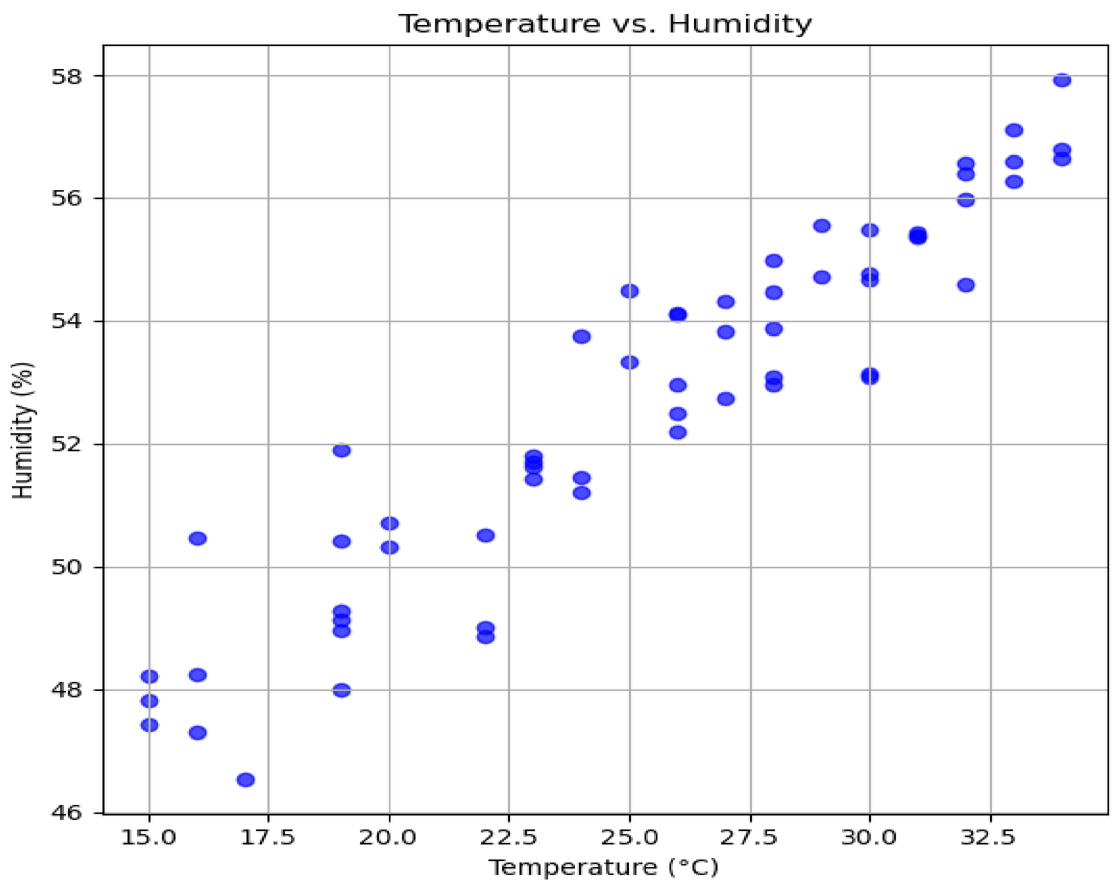
Here are some key features of scatter plots:

- Data representation: Each data point in the dataset is represented by a dot or another marker on the graph.
- Axes: The x-axis (horizontal) and y-axis (vertical) represent the two variables being studied.
- Trends and patterns: The overall pattern of the dots can reveal trends and relationships between the two variables. For example, a positive correlation means the dots will generally trend upwards from left to right, while a negative correlation would show a downward trend.
- Outliers: Scatter plots can also highlight outliers, data points that significantly deviate from the overall pattern.
- Applications: Scatter plots are used in various fields, including science, engineering, finance, and social sciences, to analyze relationships between various quantitative data points.

Program:

```
import numpy as np

import matplotlib.pyplot as plt

# Generate random temperature and humidity values with a positive correlation

np.random.seed(10)  # Set seed for reproducibility

temperatures = np.random.randint(15, 35, size=60)

humidity = 40 + temperatures * 0.5 + np.random.randn(60)  # Add some noise

plt.figure(figsize=(6, 6))  # Adjust figure size if needed
```

```
# Scatter plot with blue circles

plt.scatter(temperatures, humidity, c="blue", alpha=0.7)

# Add labels and title

plt.xlabel("Temperature (°C)")

plt.ylabel("Humidity (%)")

plt.title("Temperature vs. Humidity")

# Show the plot

plt.grid(True)

plt.tight_layout()

plt.show()
```



5a) Write a Python program to Demonstrate how to Draw a Histogram Plot using Matplotlib

What is a Histogram:

A histogram is a graphical representation of the distribution of a dataset. It displays the frequencies of different values or ranges of values in a dataset.

Why Use a Histogram:

Visualizing Distribution: Histograms provide a visual summary of the distribution of a dataset, helping to understand the central tendency, variability, and shape of the data.

Identifying Patterns: Histograms can reveal patterns and trends in the data, such as modes, clusters, or outliers. Comparison: Useful for comparing different datasets or understanding changes in a dataset over time.

How to Create a Histogram:

- Data Preparation: Collect the dataset that you want to analyze. Choose the Number of Bins:

Decide on the number of bins (intervals) that the data range will be divided into. This influences the granularity of the histogram.

- Plotting:

Use a plotting library (e.g., Matplotlib in Python) to create a histogram. Plot the data on the x-axis and the frequency (or density) on the y-axis.

- Customize:

Adjust parameters like color, transparency, bin edges, and labels to enhance visualization. Add axis labels and a title for clarity.

- Interpretation:

Analyze the histogram to understand the data distribution. Look for patterns, peaks, skewness, or any other characteristics that provide insights into the dataset.

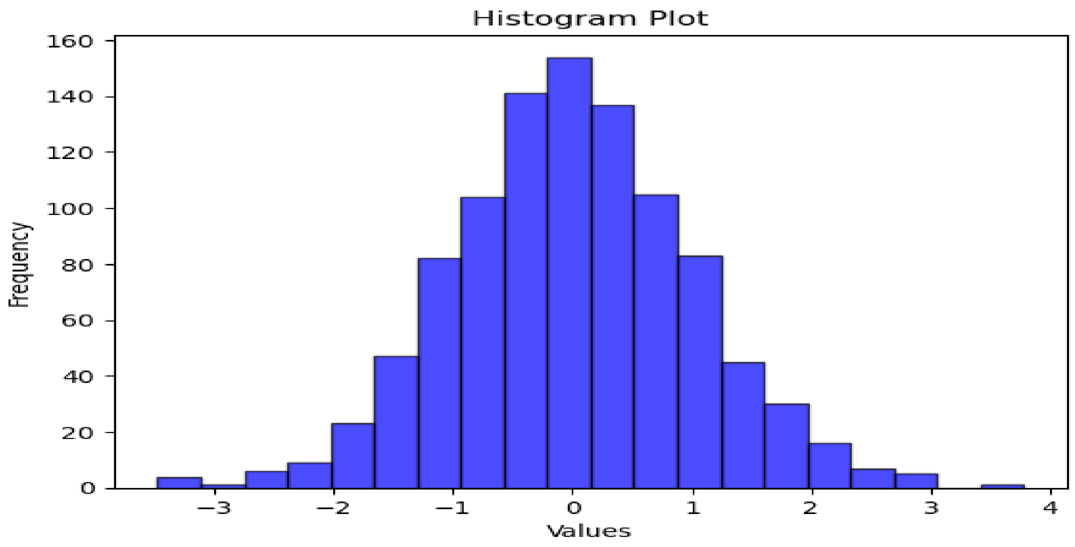
Key Components of a Histogram:

- Bins: Intervals along the x-axis that represent ranges of data values.
- Frequency: Number of data points falling into each bin.
- Axis Labels: Descriptions for the x and y axes.
- Title: A title that provides context for the histogram.
- Bars: Rectangles whose heights represent the frequency of data within each bin.

Program:

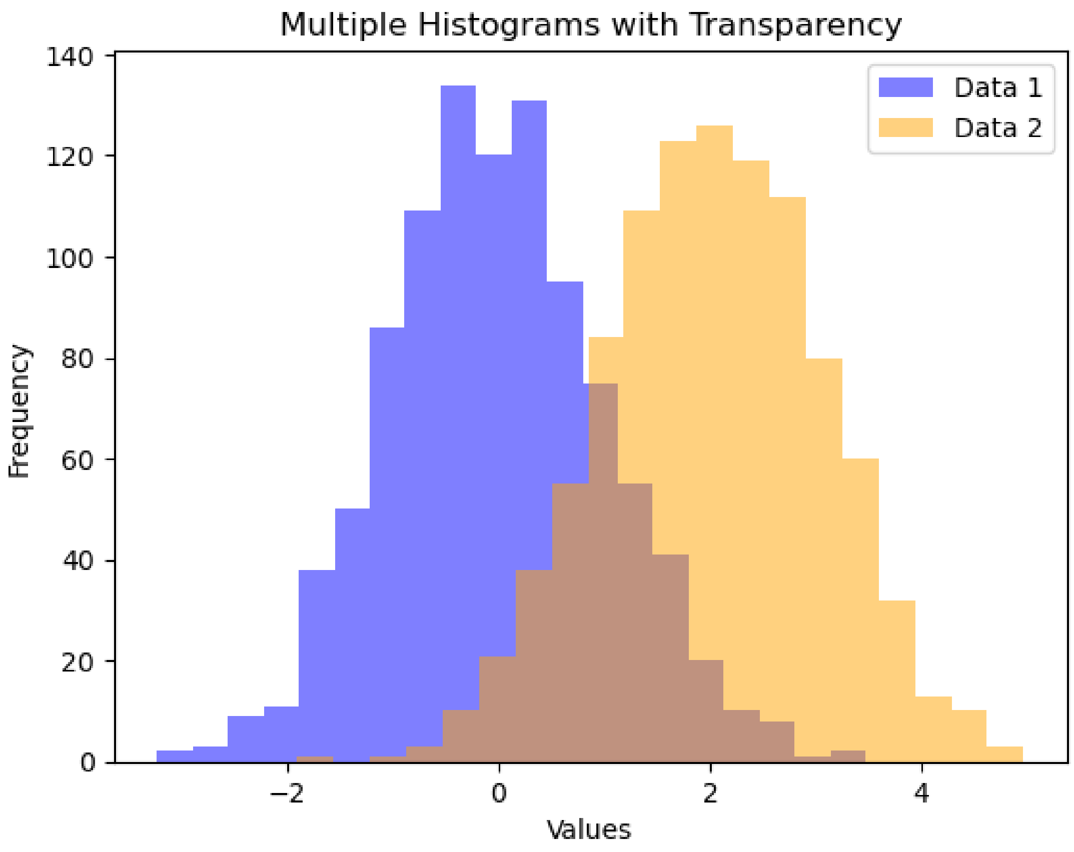
```
import matplotlib.pyplot as plt

import numpy as np

# Generate random data for demonstration

data = np.random.randn(1000)

# Create histogram

plt.hist(data, bins=20, color='blue', alpha=0.7,edgecolor='black')

# Add labels and title

plt.xlabel('Values')

plt.ylabel('Frequency')

plt.title('Histogram Plot')

# Show the plot

plt.show()
```

Output:



Variation 1: Multiple Histograms with Transparency

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

# Generate random data for demonstration
data1 = np.random.randn(1000)
data2 = np.random.randn(1000) + 2  # Shift second set for comparison

# Create histograms with transparency
plt.hist(data1, bins=20, color='blue', alpha=0.5, label='Data 1')
plt.hist(data2, bins=20, color='orange', alpha=0.5, label='Data 2')

# Add labels and title
plt.xlabel('Values')
plt.ylabel('Frequency')
plt.title('Multiple Histograms with Transparency')

# Add legend
plt.legend()

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


Multiple Histograms with Transparency
Data 1
Data 2
Frequency
Values
0
20
40
60
80
100
120
140
−2
0
2
4

5b) Write a Python program to Demonstrate how to Draw a Pie Chart using Matplotlib.

What is a Pie Chart:

A pie chart is a circular statistical graphic that is divided into slices to illustrate numerical proportions. Each slice represents a proportion of the whole, and the size of each slice corresponds to the quantity it represents. Why Use a Pie Chart:

Visualizing Proportions:

- Pie charts are effective for visually representing the distribution of parts within a whole.
- Comparisons: Easily compare the relative sizes of different categories or components.
- Simple Representation: Provide a clear and simple way to communicate percentages or proportions.

How to Create a Pie Chart:

- Data Preparation:

Prepare a dataset with values representing proportions or percentages. Extract labels and values from the dataset.

- Plotting:

Use a plotting library (e.g., Matplotlib in Python) to create a pie chart. Use the plt.pie() function, providing values, labels, and additional parameters.

- Customization:

Customize the pie chart with colors, labels, explosion, shadow, and other aesthetic options. Add a title using plt.title() for clarity.

- Show the Plot:

Display the pie chart using plt.show() or save it as an image file.

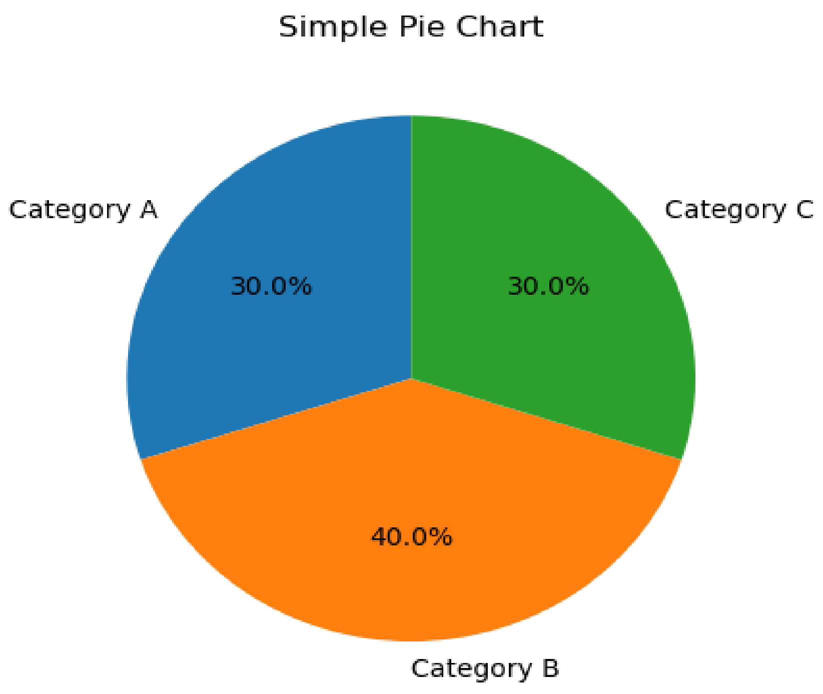
Key Components of a Pie Chart:

- Slices: Representing different categories or components.
- Labels: Descriptions for each slice.
- Proportions: The size of each slice corresponds to the proportion it represents.
- Explode: Detaching slices for emphasis.
- Shadow: Adding a shadow effect for visibility.
- Title: Providing context for the entire pie chart.

Program:

```
# Simple Pie Chart
import matplotlib.pyplot as plt

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

# Create a simple pie chart
plt.pie(values, labels=categories, autopct='%1.1f%%', startangle=90)

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

# Show the pie chart
plt.show()
```

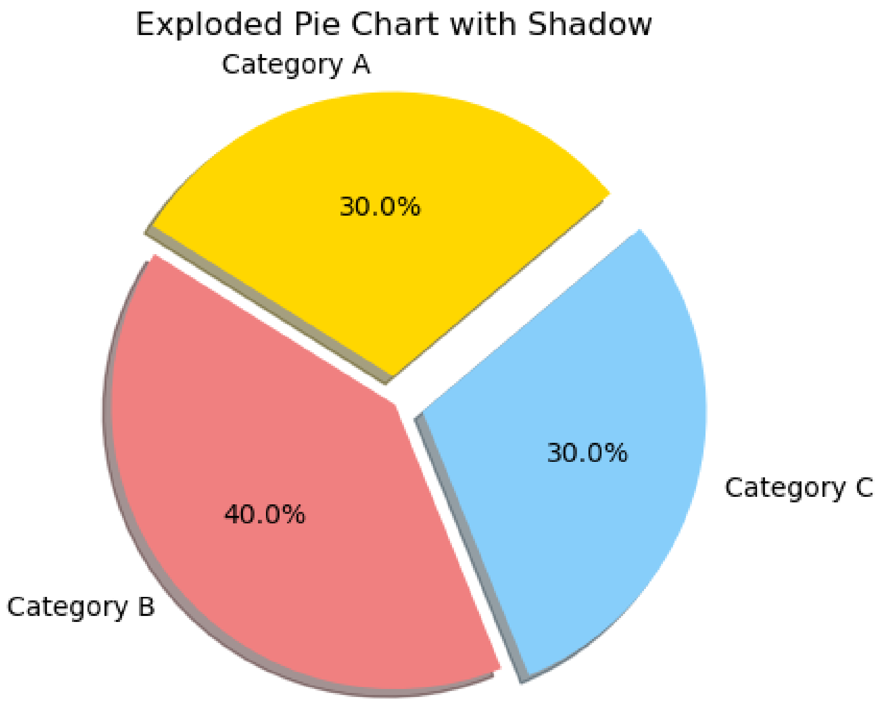
Output:



Variation 1: Exploded Pie Chart with Shadow

```
import matplotlib.pyplot as plt
# Example data
categories = ['Category A', 'Category B', 'Category C']
values = [30, 40, 30]
colors = ['gold', 'lightcoral', 'lightskyblue']
# Exploded pie chart with shadow
plt.pie(values, labels=categories, autopct='%1.1f%%', startangle=40, explode=(0.1, 0, 0.1), shadow=True,colors=colors)
plt.title('Exploded Pie Chart with Shadow')
plt.show()
```

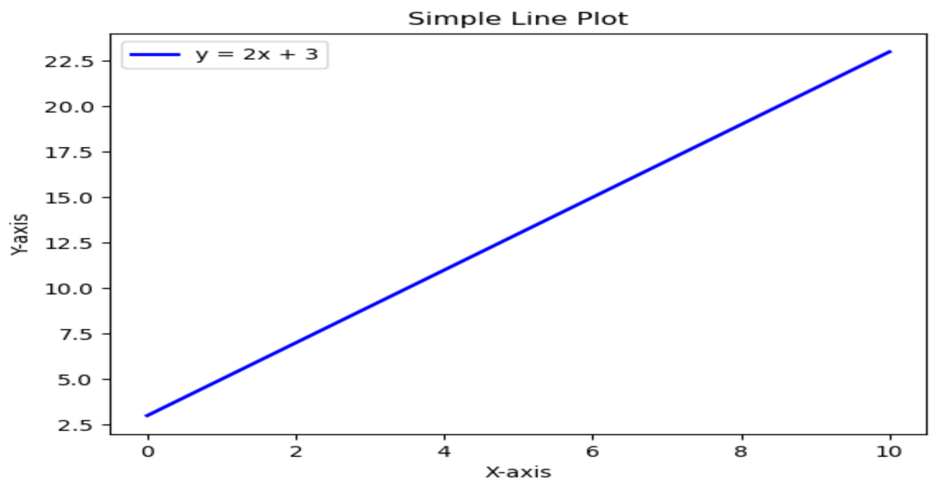
Output:


Exploded Pie Chart with Shadow
Category A
30.0%
30.0%
Category C
40.0%
Category B

6a)Write a Python program to illustrate Linear Plotting using Matplotlib

```
import matplotlib.pyplot as plt
import numpy as np
# Generate sample data
x = np.linspace(0, 10, 100)
y = 2 * x + 3
# Simple line plot
plt.plot(x, y, label='y = 2x + 3', color='blue',
        linestyle='-', linewidth=2)
# Adding labels and title
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Line Plot')
# Adding legend
plt.legend()
# Show the plot
plt.show()
```

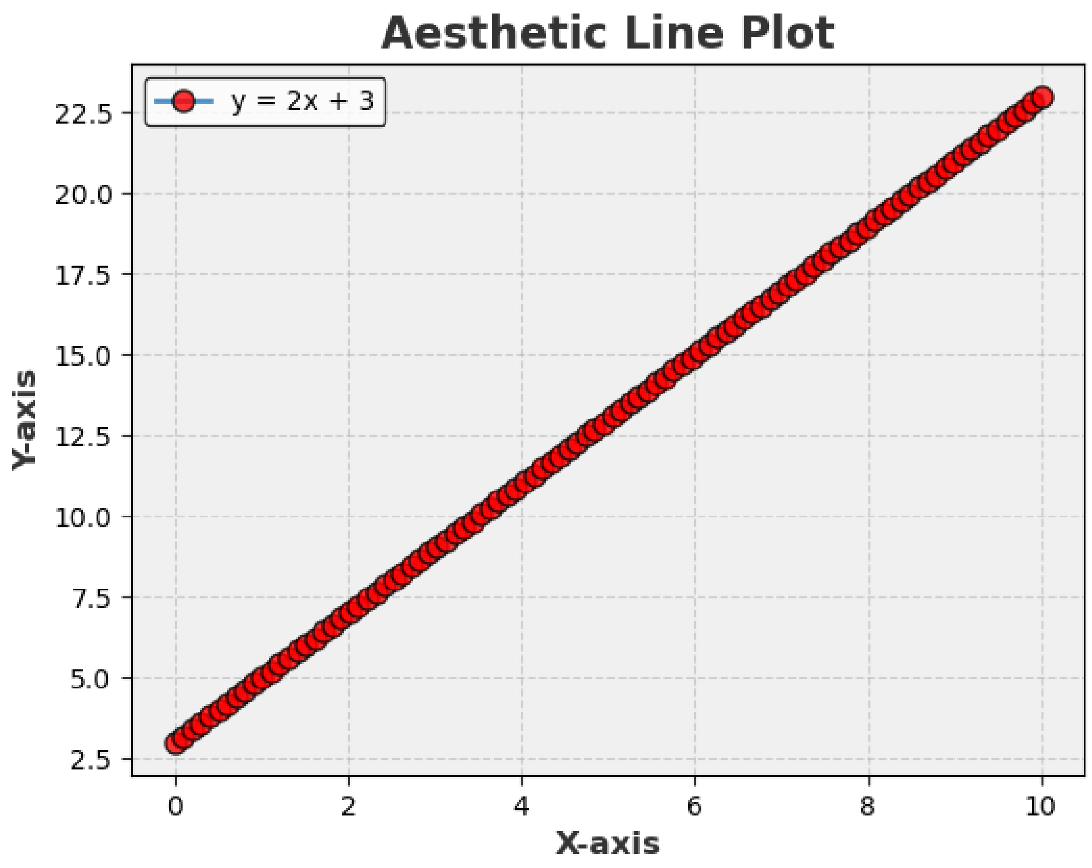
Output:



6b) Write a Python program to illustrate liner plotting with line formatting using Matplotlib.

In [1]:

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

# Generate sample data
x = np.linspace(0, 10, 100)
y = 2 * x + 3

# Create a more aesthetic line plot
plt.plot(x, y, label='y = 2x + 3',
        color='#1f77b4', linestyle='-',
        linewidth=2, marker='o',
        markersize=8, markerfacecolor='red',
        markeredgecolor='black', alpha=0.8)

# Adding labels and title with custom font settings
plt.xlabel('X-axis', fontsize=12,
          fontweight='bold', color='#333333')
plt.ylabel('Y-axis', fontsize=12,
          fontweight='bold', color='#333333')
plt.title('Aesthetic Line Plot', fontsize=16,
         fontweight='bold', color='#333333')
# Adjusting grid style
plt.grid(True, linestyle='--', alpha=0.5)

# Adding legend with custom settings
plt.legend(loc='upper left', fontsize=10, fancybox=True, framealpha=0.8, edgec
olor='black')

# Customizing tick parameters
plt.tick_params(axis='both', labelsize=10, color='#333333')

# Adding a background color to the plot
plt.gca().set_facecolor('#f0f0f0')

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



In [ ]:

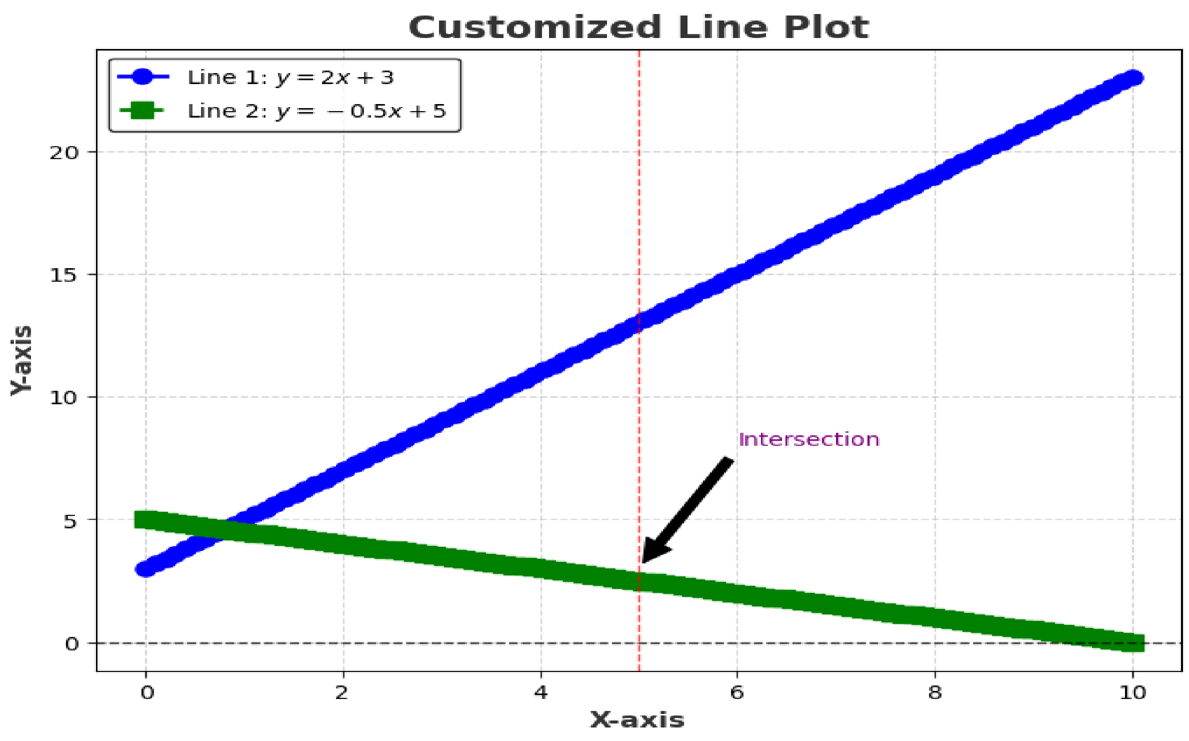
Variation 1:

```
import matplotlib.pyplot as plt
import numpy as np
# Generate sample data
x = np.linspace(0, 10, 100)
y1 = 2 * x + 3
y2 = -0.5 * x + 5

# Customizing the Line Plot
plt.figure(figsize=(8, 6))  # Set the figure size

# Plotting multiple lines with different styles
```

```
plt.plot(x, y1, label='Line 1: $y = 2x + 3$',
        color='blue', linestyle='-',
        linewidth=2, marker='o',
        markersize=8)
plt.plot(x, y2, label='Line 2: $y = -0.5x + 5$',
        color='green', linestyle='--',
        linewidth=2, marker='s', markersize=8)

# Adding labels and title with custom font settings
plt.xlabel('X-axis', fontsize=12, fontweight='bold', color='#333333')
plt.ylabel('Y-axis', fontsize=12, fontweight='bold', color='#333333')
plt.title('Customized Line Plot', fontsize=16, fontweight='bold', color='#333333')

# Adjusting grid style
plt.grid(True, linestyle='--', alpha=0.5)

# Adding legend with custom settings
plt.legend(loc='upper left', fontsize=10,
          fancybox=True, framealpha=0.8,
          edgecolor='black')

# Customizing tick parameters
plt.tick_params(axis='both', labelsize=10,
               color='#333333')

# Adding horizontal line at y=0
plt.axhline(y=0, color='black', linestyle='--',
           linewidth=1, alpha=0.7)
```

```
# Adding vertical line at x=5
plt.axvline(x=5, color='red', linestyle='--',
            linewidth=1, alpha=0.7)

# Adding annotations
plt.annotate('Intersection', xy=(5, 3),
             xytext=(6, 8),
             arrowprops=dict(facecolor='black',
                             shrink=0.05),
             fontsize=10, color='purple')

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

Output:



7.Write a Python program which explains uses of customizing seaborn plots with Aesthetic functions.

In Seaborn, aesthetic functions are the functions that allow you to control the visual appearance of plots. These functions help you customize the style, context, color palette, and more. Some of the aesthetic functions in Seaborn include:

-sns.set_style(): Sets the aesthetic style of the plots. -sns.set_context(): Sets the context parameters for the plots. -sns.set_palette(): Sets the color palette for the plots. -sns.color_palette(): Generates color palettes. -sns.set() : Sets the aesthetic parameters in one step. -sns.despine(): Removes the top and right spines from plots. -sns.set_theme(): Sets the visual theme for Seaborn.

These functions allow you to create plots with different visual styles, color schemes, and overall appearances, making it easier to customize your visualizations according to your preferences or the requirements of your analysis.

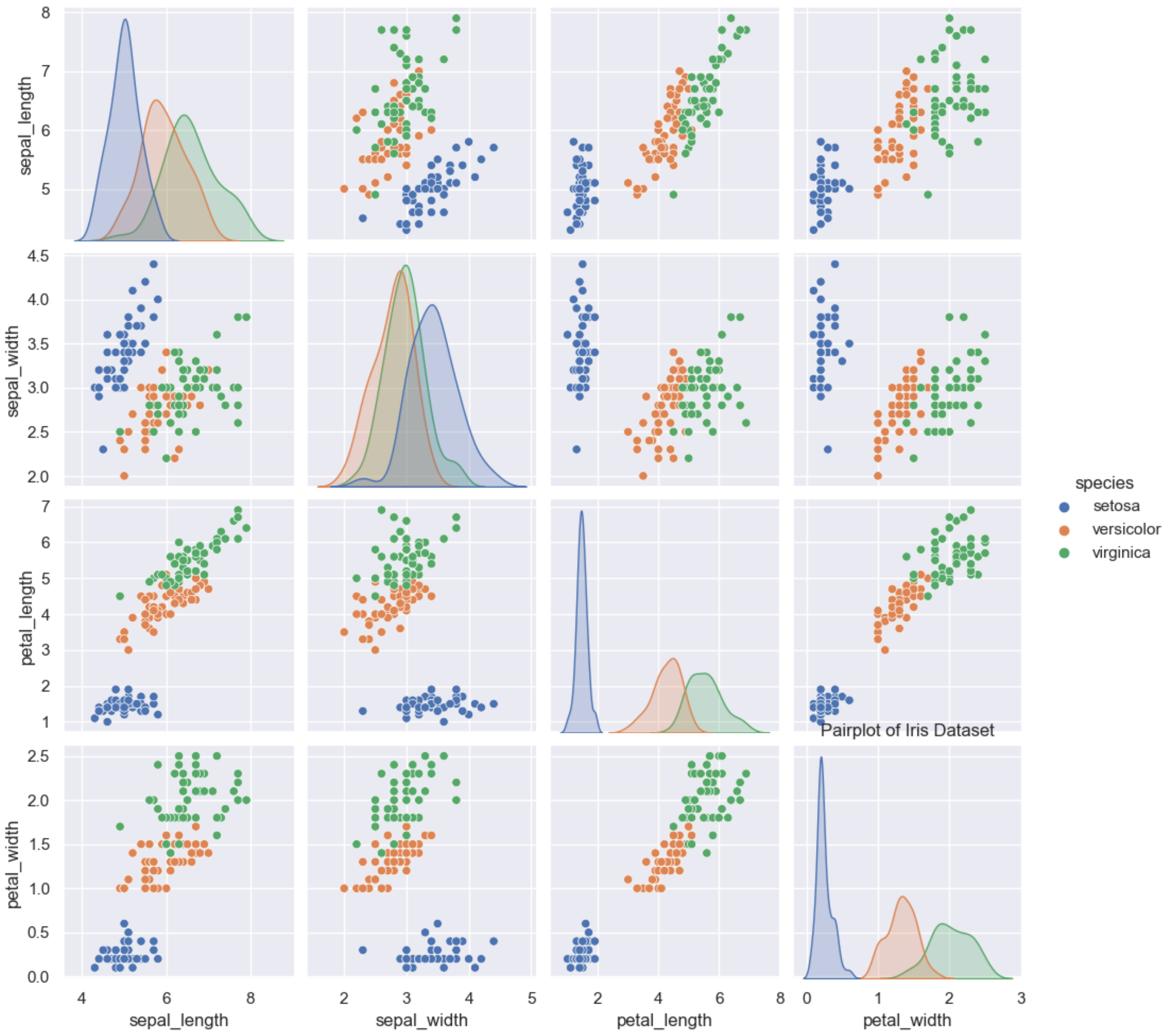
Program:

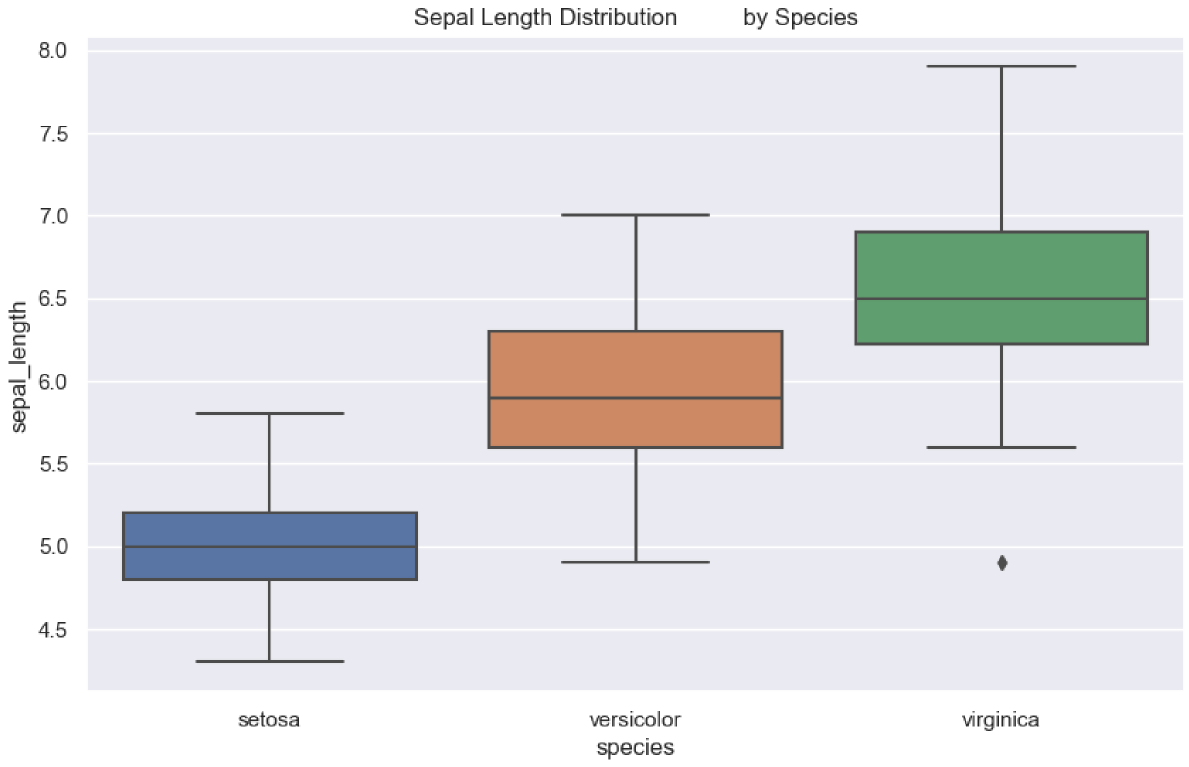
```
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd


# Load the Iris dataset
iris = sns.load_dataset("iris")

# Display the first few rows of the dataset
#print(iris.head())

# Summary statistics
print(iris.describe())

# Pairplot to visualize relationships between variables
sns.pairplot(iris, hue="species")
plt.title('Pairplot of Iris Dataset')
plt.show()

# Boxplot to visualize the distribution of each feature by species
plt.figure(figsize=(10, 6))
```

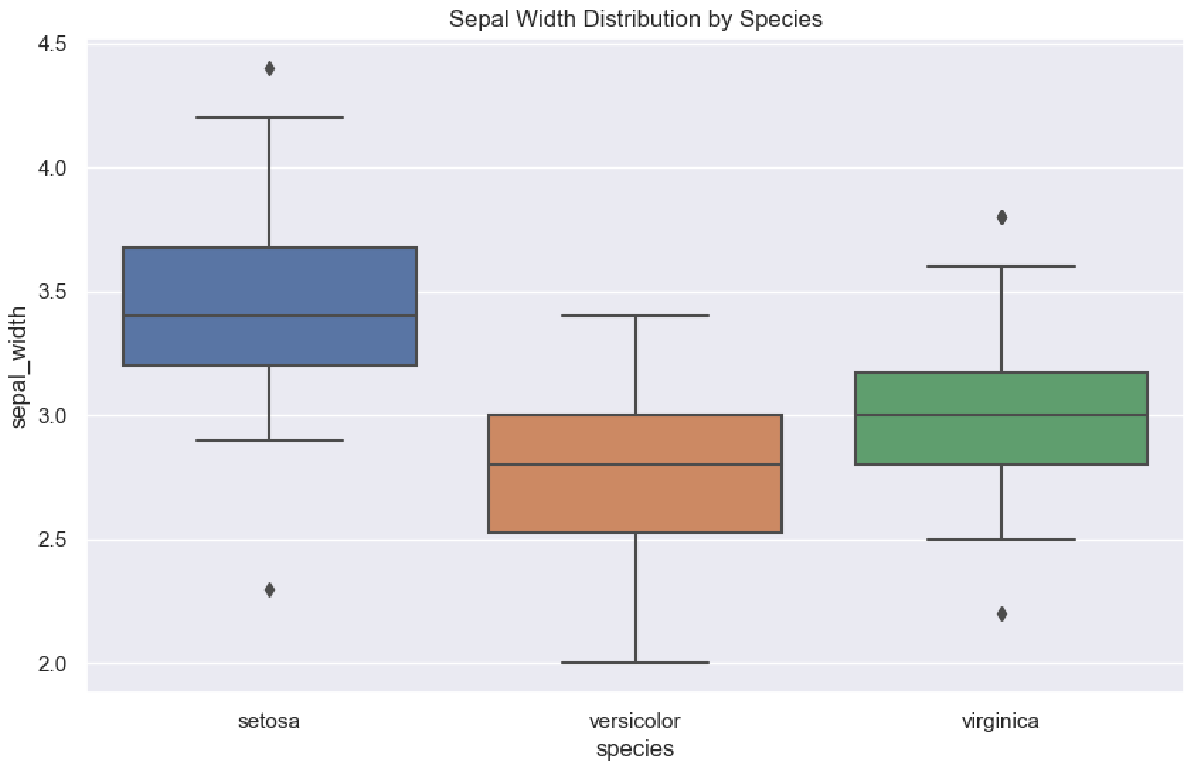
```
sns.boxplot(x="species",
            y="sepal_length", data=iris)
plt.title('Sepal Length Distribution          by Species')
plt.show()

plt.figure(figsize=(10, 6))
sns.boxplot(x="species", y="sepal_width",
            data=iris)
plt.title('Sepal Width Distribution by Species')
plt.show()

plt.figure(figsize=(10, 6))
sns.boxplot(x="species", y="petal_length", data=iris)
plt.title('Petal Length Distribution by Species')
plt.show()

plt.figure(figsize=(10, 6))
sns.boxplot(x="species", y="petal_width", data=iris)
plt.title('Petal Width Distribution by Species')
plt.show()

# Correlation heatmap
plt.figure(figsize=(8, 6))
sns.heatmap(iris.corr(), annot=True,
            cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Heatmap of Iris Dataset')
plt.show()
```

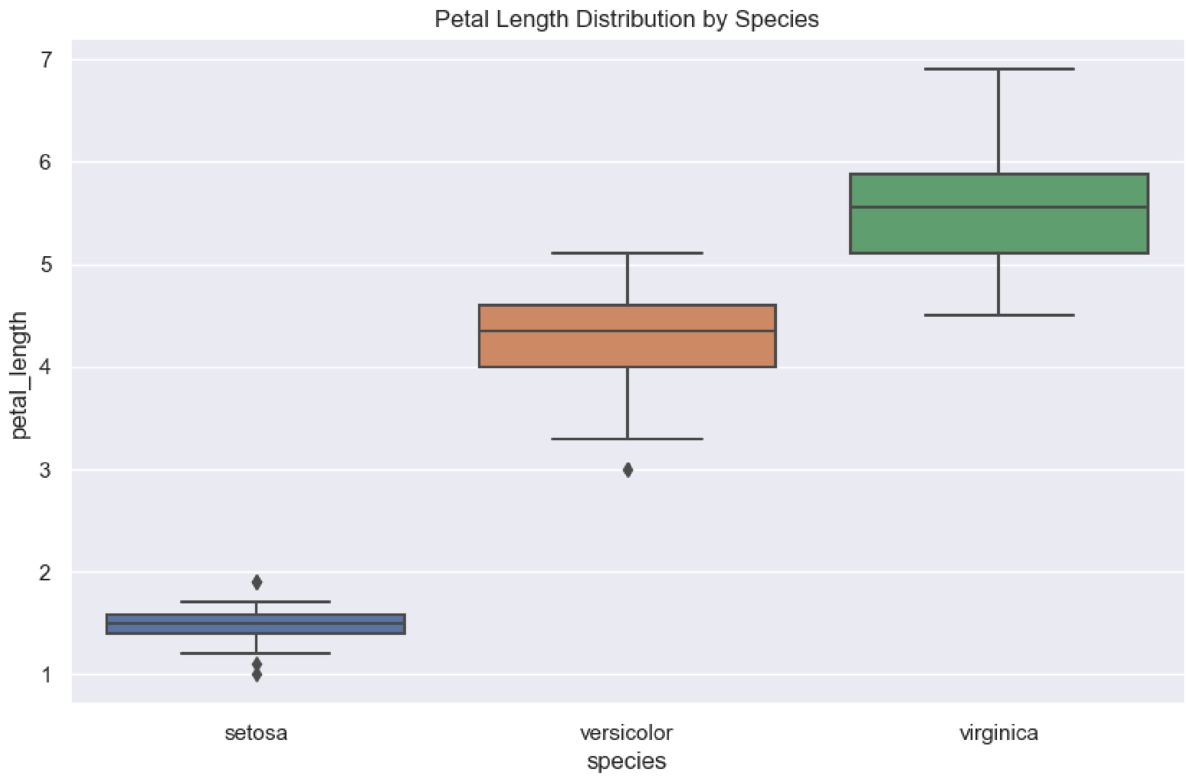
Output:

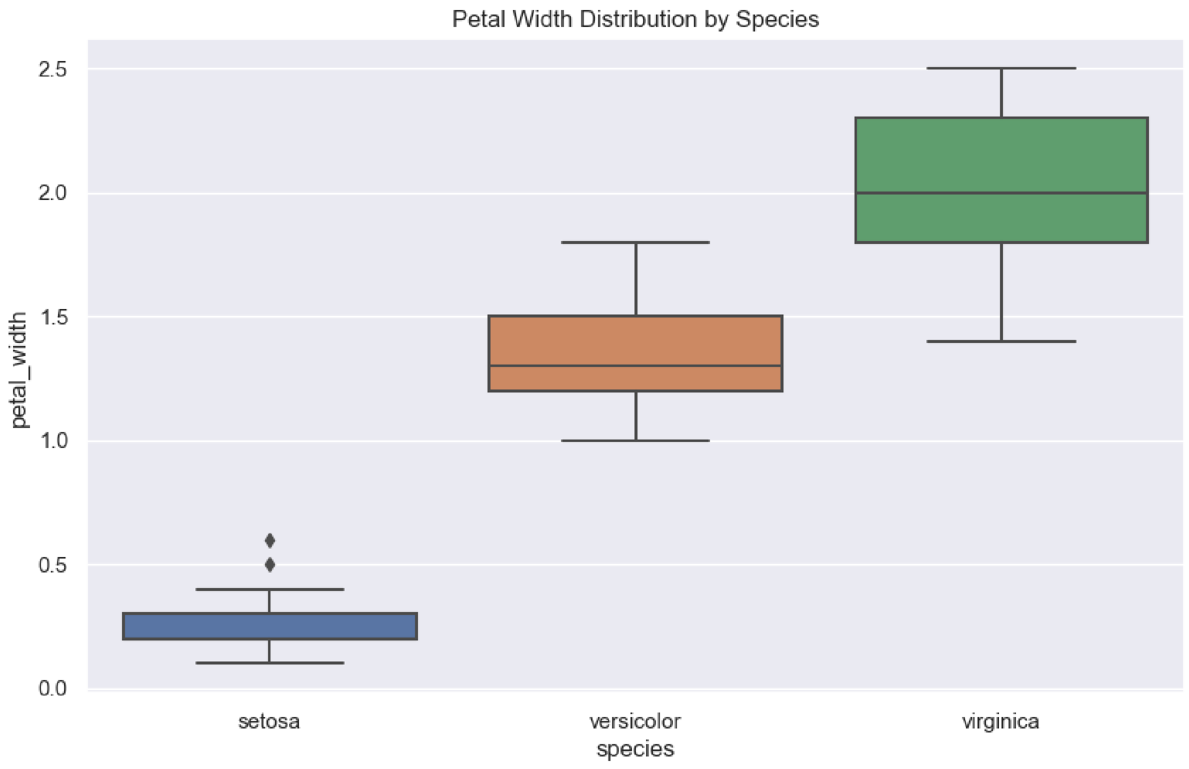
```
       sepal_length  sepal_width  petal_length  petal_width
count    150.000000   150.000000    150.000000   150.000000
mean       5.843333     3.057333      3.758000     1.199333
std        0.828066     0.435866      1.765298     0.762238
min        4.300000     2.000000      1.000000     0.100000
25%        5.100000     2.800000      1.600000     0.300000
50%        5.800000     3.000000      4.350000     1.300000
75%        6.400000     3.300000      5.100000     1.800000
max        7.900000     4.400000      6.900000     2.500000
```

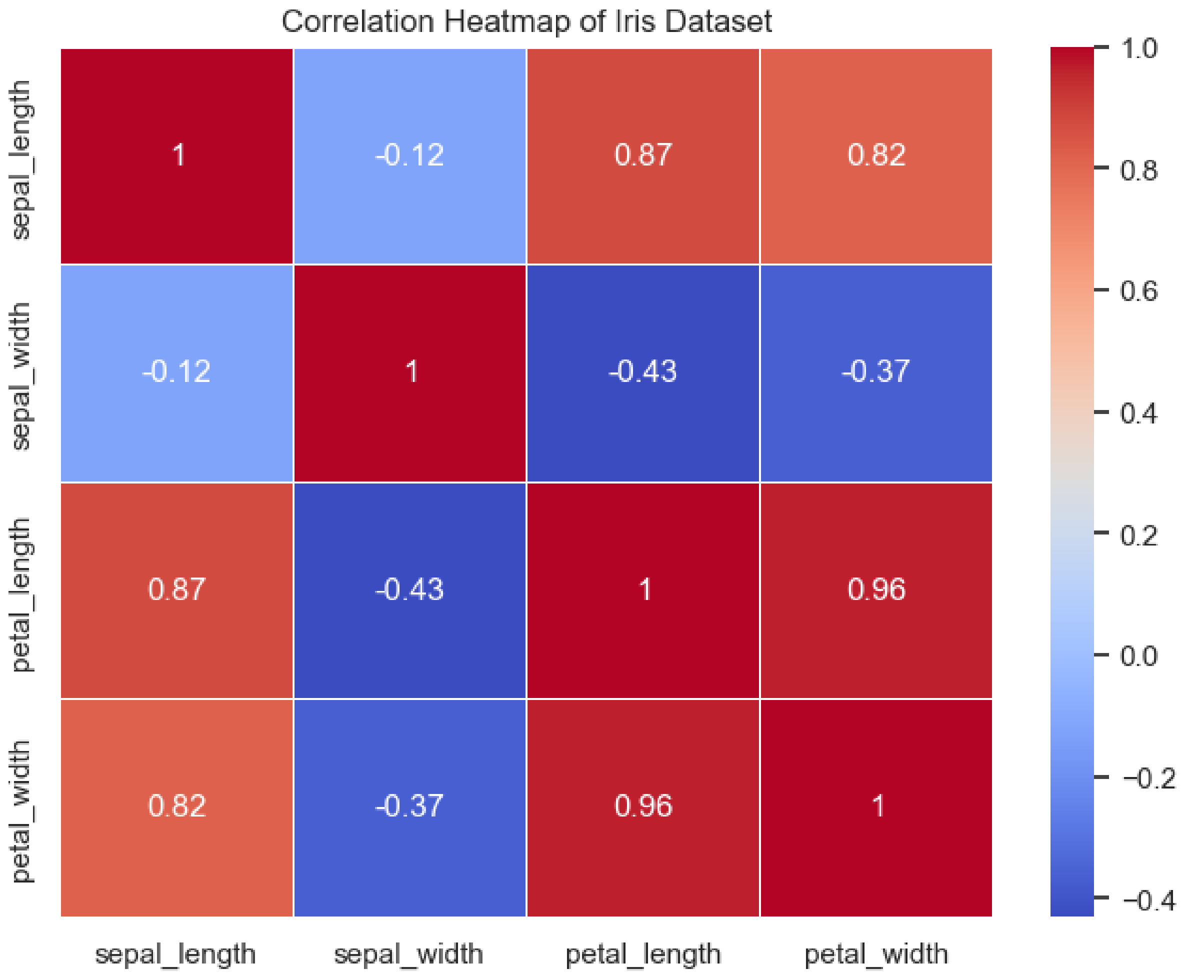

sepal_length
sepal_width
petal_length
petal_width
species
setosa
versicolor
virginica
Pairplot of Iris Dataset


Sepal Length Distribution by Species
sepal_length
setosa
versicolor
virginica
species


Sepal Width Distribution by Species
4.5
4.0
3.5
3.0
2.5
2.0
sepal_width
setosa
versicolor
virginica
species


Petal Length Distribution by Species
7
6
5
4
3
2
1
petal_length
setosa
versicolor
virginica
species





	sepal_length	sepal_width	petal_length	petal_width
sepal_length	1	-0.12	0.87	0.82
sepal_width	-0.12	1	-0.43	-0.37
petal_length	0.87	-0.43	1	0.96
petal_width	0.82	-0.37	0.96	1

8.Write a Python program to explain working with bokeh line graph using Annotations and Legends.

a) Write a Python program for plotting different types of plots using Bokeh.

```
from bokeh.plotting import figure, show
from bokeh.models import Label, Legend
from bokeh.io import output_file

# Create a figure
p = figure(title="Bokeh Line  Graph with Annotations          and Legends",
           x_axis_label="X-axis",
           y_axis_label="Y-axis")

# Generate some sample data
x_values = [1, 2, 3, 4, 5]
y1_values = [5, 3, 4, 2, 6]
y2_values = [1, 2, 0, 3, 4]

# Plot lines on the figure
line1 = p.line(x_values,
               y1_values,
               line_width=2,
               legend_label="Line 1",
               line_color="blue")
line2 = p.line(x_values,
               y2_values,
               line_width=2,
               legend_label="Line 2",
               line_color="green")

# Add annotations to the graph
annotation1 = Label(x=3, y=4,
                    text="Annotation 1",
                    text_font_size="10pt",
                    text_color="black")
annotation2 = Label(x=2, y=2,
                    text="Annotation 2",
                    text_font_size="10pt",
                    text_color="black")

p.add_layout(annotation1)
p.add_layout(annotation2)

# Add legends to the graph
legend = Legend(items=[("Line 1", [line1]),
                       ("Line 2", [line2])],
                location="top_left")
p.add_layout(legend)

# Save the output to an HTML file
output_file("bokeh_line_graph.html")

# Show the plot
show(p)
```

9. Write a Python program to draw 3D Plots using Plotly Libraries.

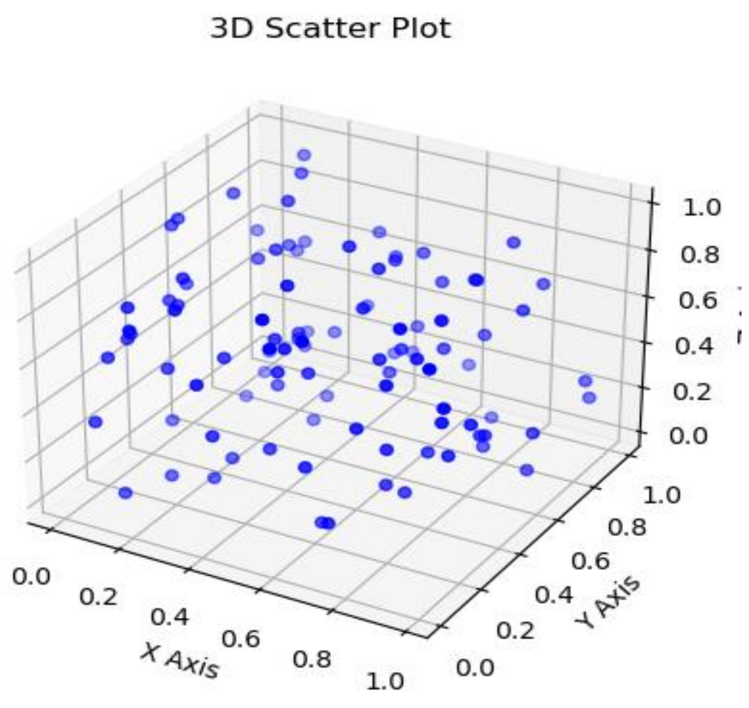
1. 3D Scatter Plot Use Case: Visualizing the distribution of points in three-dimensional space. Examining the relationships and patterns among three variables. Significance: Useful for understanding the spatial arrangement of data points. Enables the identification of clusters or trends in three-dimensional datasets.
2. 3D Line Plot Use Case: Representing a curve or trajectory in three-dimensional space.> Analyzing the behavior of a variable over a continuous range. Significance: Ideal for visualizing 3D paths or spirals. Provides insights into the movement or variation of a variable along three axes.
3. 3D Surface Plot Use Case: Representing a surface defined by two independent variables and their dependent variable. Visualizing functions of two variables. Significance: Useful for understanding the behavior of a surface in a three-dimensional space. Commonly used in mathematical and scientific visualizations.
4. 3D Bar Plot Use Case: Visualizing three-dimensional data with categorical variables. Representing volumes associated with different categories. Significance: Provides a clear representation of the distribution of values across two categorical variables. Useful for comparing quantities associated with different categories in a 3D space.
5. 3D Contour Plot Use Case: Visualizing the contours of a three-dimensional surface. Identifying regions of similar values in three-dimensional data. Significance: Useful for understanding the structure of a surface and variations in different regions. Commonly employed in scientific and engineering applications.
6. 3D Quiver Plot Use Case: Representing vector fields in three-dimensional space. Visualizing the direction and magnitude of vectors at different points. Significance: Useful for studying fluid dynamics, electromagnetic fields, or any situation involving vectors. Provides insights into the spatial distribution of vector quantities. When to Use 3D Plots:

Use 3D plots when exploring relationships involving three variables. Consider 3D plots when traditional 2D plots may not effectively convey the information. Ensure that 3D plots enhance rather than complicate the understanding of your data. Note: While 3D plots can be visually appealing, they should be used judiciously. In some cases, too much complexity may hinder interpretation, and alternative visualization methods might be more suitable.

Variation 1: 3D Scatter Plot

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

# Create random data
np.random.seed(42)
n_points = 100
x = np.random.rand(n_points)
y = np.random.rand(n_points)
z = np.random.rand(n_points)

# Create 3D scatter plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(x, y, z, c='blue', marker='o')

# Add labels and title
ax.set_xlabel('X Axis')
ax.set_ylabel('Y Axis')
ax.set_zlabel('Z Axis')
ax.set_title('3D Scatter Plot')
# Show the plot
plt.show()
```

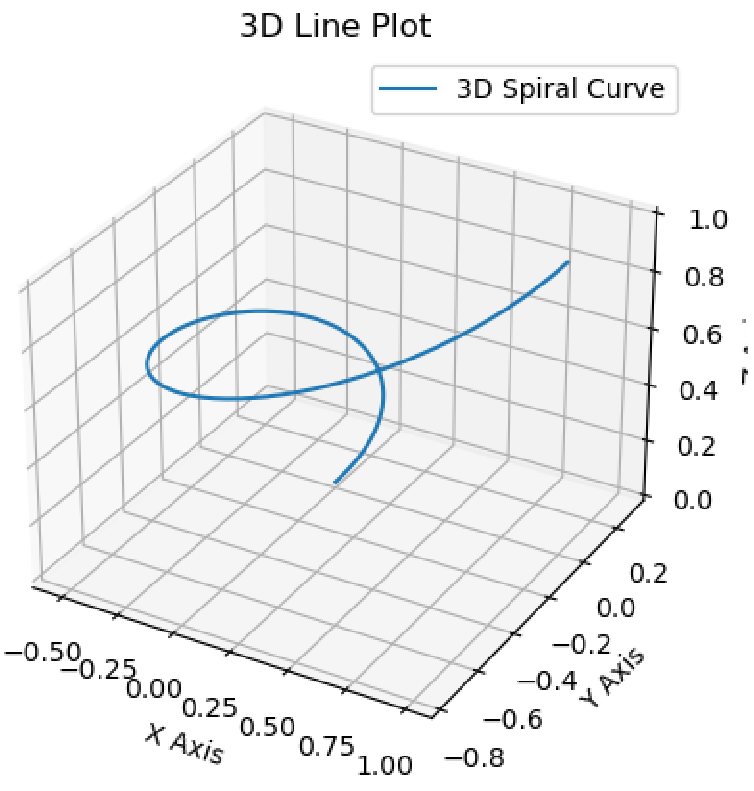
Output:



Variation 2: 3D Line Plot

In [2]:

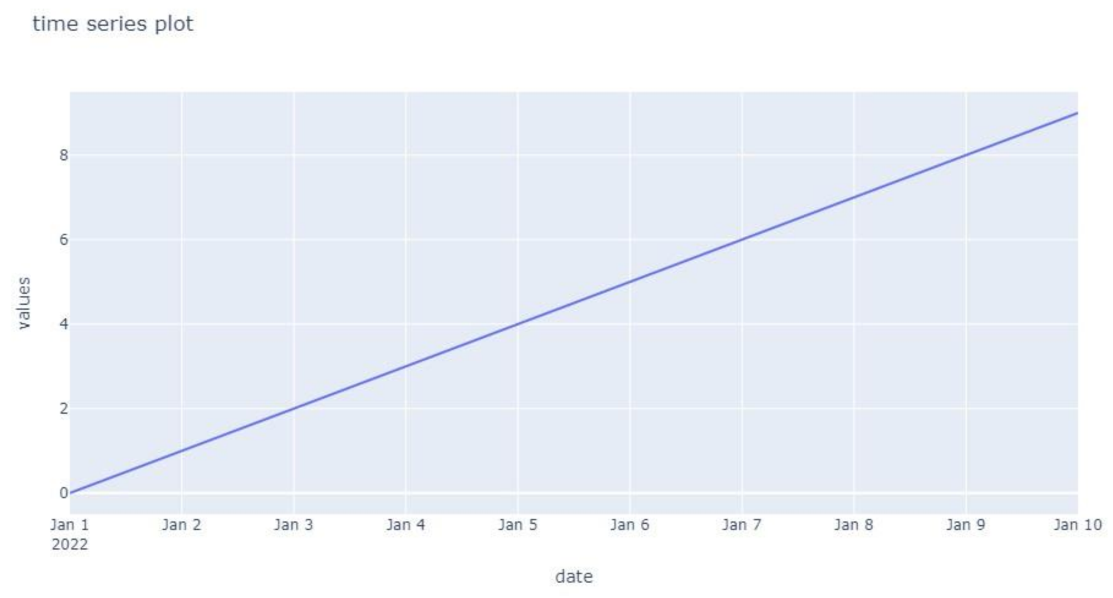
```
import matplotlib.pyplot as plt
import numpy as np

# Create random data
theta = np.linspace(0, 2*np.pi, 100)
z = np.linspace(0, 1, 100)
# Create 3D line plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot(z * np.cos(theta), z * np.sin(theta),
        z,
        label='3D Spiral Curve')
# Add labels and title
ax.set_xlabel('X Axis')
ax.set_ylabel('Y Axis')
ax.set_zlabel('Z Axis')
ax.set_title('3D Line Plot')

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



10a) write a python program to draw a time series using plotly library

```
import plotly.graph_objects as go

import pandas as pd

date_rng=pd.date_range(start='2022-01-01',end='2022-01-10',freq='D')

time_series_data=pd.Series(range(len(date_rng)),index=date_rng)

fig=go.Figure()

fig.add_trace(go.Scatter(x=time_series_data.index,y=time_series_data.values,mode='lines',name='time series'))

fig.update_layout(title='time series plot',xaxis_title='date',yaxis_title='values')

fig.show()
```

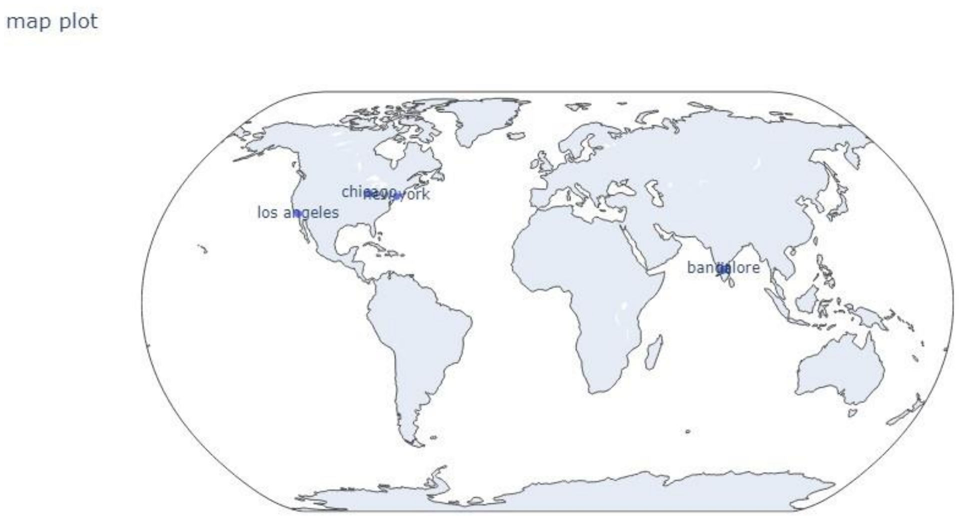
Output:



10.b) Write a Python program for creating Maps using Plotly Libraries.

```
import plotly.express as px

# Create a sample dataframe with location data

data = {'City': ['New York', 'Los Angeles', 'Chicago', 'Houston',

           'Phoenix'],

     'Lat': [40.7128, 34.0522, 41.8781, 29.7604, 33.4484],

     'Lon': [-74.0060, -118.2437, -87.6298, -95.3698, -112.0740]}

df = pd.DataFrame(data)

# Create a map plot

fig = px.scatter_geo(df, lat='Lat', lon='Lon', text='City',

             title='Map Plot using Plotly',

             projection='natural earth')

fig.show()
```

Output:


map plot
chicago
new york
los angeles
bangalore