***N.B.K.R Institute of Science & Technology***

***AUTONOMOUS***

***Vidyanagar***

***CERTIFICATE***

***Certified that this is a bona fide record of practical work donein the…………………………………………………………………… Laboratory by Mr./Ms.…………………………………………………. Roll No. ……………………… during the year ……………….........***

***Number of Experiments done:***

***Staff Member in-charge:***

***Examiner:***

# *INDEX*

# EXPERIMENT-1

**Aim:** Download and install any Python-Programming Environment and install basic packages

(numpy, pandas, matplotlib, scikitlearn, etc).

## Description:

**1. NumPy**

NumPy (Numerical Python) is the fundamental package for scientific computing in Python. It provides support for arrays, matrices, and high-level mathematical functions to operate on them efficiently. NumPy allows you to work with large datasets more efficiently than native Python lists and enables element-wise computations on arrays. Its powerful n-dimensional array objects make it ideal for working with linear algebra, Fourier transforms, and random number generation. NumPy also serves as the backbone for other scientific libraries such as Pandas and SciPy.

**Command to Install:** pip install numpy

**2. Pandas**

Pandas is a powerful and flexible data manipulation and analysis library built on top of NumPy. It introduces two primary data structures: Series (1D) and DataFrame (2D), which allow for easy manipulation of structured data like CSV files, Excel spreadsheets, and SQL tables. With Pandas, users can easily handle missing data, reshape datasets, and perform group operations (e.g., group by, pivot tables). It’s a go-to library for anyone working with data cleaning, analysis, or visualization.

**Command to Install:** pip install pandas

**3. Matplotlib**

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It’s highly customizable and can produce a wide variety of plots and charts, including line plots, histograms, bar charts, and scatter plots. It integrates well with NumPy, Pandas, and other libraries, allowing users to plot data directly from DataFrames. Matplotlib is ideal for creating publication-quality visualizations and is a favorite in data analysis, machine learning, and scientific research.

**Command to Install:** pip install matplotlib

**4. Scikit-learn**

Scikit-learn is a popular machine learning library built on NumPy, SciPy, and Matplotlib. It provides simple and efficient tools for data mining and data analysis. Scikit-learn includes a wide array of machine learning algorithms, from supervised learning (classification and regression) to unsupervised learning (clustering, dimensionality reduction). It also includes utilities for model selection, data preprocessing, and model evaluation. It’s widely used in academic research and production environments alike due to its ease of use and powerful performance.

**Command to Install:** pip install scikit-learn

**Result:** The above packages has been successfully installed.

EXP.NO: 2 DATE:

## EXPERIMENT-2

**Aim:** Develop python programs using numpy module

**Description:** NumPy is a powerful Python library essential for numerical computing, providing support for large, multi-dimensional arrays and matrices, along with a vast collection of mathematical functions to operate on these data structures. It serves as the ultimate foundation for many other scientific libraries, such as SciPy and Pandas. Key features include efficient array operations, broadcasting capabilities, and extensive mathematical functions, making it ideal for data analysis, machine learning, and scientific research. With its ability to handle complex data, NumPy is a go-to tool for developers and researchers alike.

**Source Code :**

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print("1D Array:", arr)
```

**Output:**

```
1D Array: [1 2 3 4 5]
```

**Source Code :**

```
import numpy as np
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
print("2D Array:\n", arr_2d)
```

**Output:**

```
2D Array:
[[1 2 3]
 [4 5 6]]
```

**Source Code :**

```
import numpy as np
arr = np.array([3, 5, 1, 8, 2])
print("Max:", np.max(arr))
```

**Output:**

```
Max: 8
```

**Source Code :**

```
import numpy as np

arr = np.array([3, 5, 1, 8, 2])

print("Min:", np.min(arr))
```

**Output:**

```
Min: 1
```

**Source Code :**

```
import numpy as np
array_a = np.array([1, 2, 3])
array_b = np.array([4, 5, 6])
addition_result = array_a + array_b
print("Addition Result:", addition_result)
```

**Output:**

```
Addition Result: [5 7 9]
```

**Result:** The above program has been successfully executed.

# EXPERIMENT-3

**Aim:** Develop python matrix programs using numpy module.

**Description:** Numpy is a powerful library in Python that facilitates efficient numerical computations, particularly with multi-dimensional arrays and matrices. It provides a range of functions for creating, manipulating, and performing operations on matrices, making it an essential tool for scientific computing, data analysis, and machine learning. Using Numpy, you can easily create matrices, perform element-wise operations, apply linear algebra techniques, and leverage broadcasting for efficient computations. Below, I'll provide some example programs showcasing various matrix operations using Numpy.

## Source Code:

```
import numpy as np
# Define two matrices
A = np.array([[3, 2, 1],
              [6, 5, 4]])
B = np.array([[1, 2, 3],
              [4, 5, 6]])
# Subtract the matrices
result_subtraction = A - B
print("Subtraction of A and B:\n", result_subtraction)
```

## Output:

```
Subtraction of A and B:
 [[ 2  0 -2]
 [ 2  0 -2]]
```

## Source Code:

```
import numpy as np

# Define two matrices
A = np.array([[1, 2],
              [3, 4]])

B = np.array([[5, 6],
              [7, 8]])

# Multiply the matrices
result_multiplication = np.dot(A, B)
```

```
print("Multiplication of A and B:\n", result_multiplication)
```

**Output:**

```
Multiplication of A and B:
 [[19 22]
 [43 50]]
```

**Source Code:**

```
import numpy as np
# Define a matrix
A = np.array([[1, 2, 3],
              [4, 5, 6]])
# Transpose the matrix
result_transpose = A.T
print("Transpose of A:\n", result_transpose)
```

**Output:**

```
Transpose of A:
 [[1 4]
 [2 5]
 [3 6]]
```

**Source Code:**

```
import numpy as np
# Define a square matrix
A = np.array([[4, 7],
              [2, 6]])
# Calculate the inverse of the matrix
result_inverse = np.linalg.inv(A)
print("Inverse of A:\n", result_inverse)
```

**Output:**

```
Inverse of A:
 [[ 0.6 -0.7]
 [-0.2  0.4]]
```

**Source Code:**

```
import numpy as np

# Define a square matrix

A = np.array([[1, 2, 3],

              [4, 5, 6],

              [7, 8, 9]])

# Calculate the trace of the matrix

result_trace = np.trace(A)

print("Trace of A:", result_trace)
```

**Output:**

```
Trace of A: 15
```

**Result:** The above program has been successfully executed.

# EXPERIMENT-4

**Aim:-** Develop a python program for reshaping of numpy arrays.

**Description:-** This Python program demonstrates how to reshape NumPy arrays efficiently. It starts by creating a 1D array of integers and then reshapes it into various 2D configurations. The program showcases the use of the **reshape()** method, allowing for flexible data manipulation. Finally, it prints both the original and reshaped arrays for comparison.

## Source code:-

```
import numpy as np
# Create a 1D NumPy array with values from 0 to 11
original_array = np.arange(12)
print("Original Array:")
print(original_array)
```

## Output:-

```
Original Array:
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

## Source code:-

```
# Reshape the array to 2D (3 rows, 4 columns)
reshaped_array_1 = original_array.reshape(3, 4)
print("\nReshaped Array (3x4):")
print(reshaped_array_1)
```

## Output:-

```
Reshaped Array (3x4):
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

## Source code:-

```
# Reshape the array to 3D (2 layers, 3 rows, 2 columns)
reshaped_array_2 = original_array.reshape(2, 3, 2)
print("\nReshaped Array (2 layers, 3 rows, 2 columns):")
print(reshaped_array_2)
```

## Output:-

```
Reshaped Array (2 layers, 3 rows, 2 columns):
[[[ 0  1]
  [ 2  3]
  [ 4  5]]

 [[ 6  7]
  [ 8  9]
  [10 11]]]
```

## Source code:-

```
# Demonstrate reshaping with -1 to automatically infer the dimension
reshaped_array_3 = original_array.reshape(-1, 6)  # Here, -1 lets NumPy determine the number of
rows
print("\nReshaped Array with -1 (Auto-inferred rows):")
print(reshaped_array_3)
```

## Output:-

```
Reshaped Array with -1 (Auto-inferred rows):
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
```

## Source code:-

```
# Confirming the shapes
print("\nShapes of the arrays:")
print(f"Original Array Shape: {original_array.shape}")
print(f"Reshaped Array (3x4) Shape: {reshaped_array_1.shape}")
print(f"Reshaped Array (2x3x2) Shape: {reshaped_array_2.shape}")
print(f"Reshaped Array with -1 Shape: {reshaped_array_3.shape}")
```

## Output:-

```
Shapes of the arrays:
Original Array Shape: (12,)
Reshaped Array (3x4) Shape: (3, 4)
Reshaped Array (2x3x2) Shape: (2, 3, 2)
Reshaped Array with -1 Shape: (2, 6)
```

**Result:-**The above program has been executed successfully.

# EXPERIMENT-5

**Aim:-** Develop dataframe objects.

**Description:-** DataFrame objects are tabular data structures that organize data into rows and columns, similar to spreadsheets or SQL tables. They allow for efficient data manipulation and analysis, supporting operations like filtering, aggregation, and joining datasets. DataFrames can handle various data types, manage missing values, and provide built-in functions for complex calculations. They are widely used in data analysis and machine learning for their versatility and ease of use.

**Source code:-**

```
import pandas as pd

data = {
    'Product': ['A', 'B', 'C'],
    'Price': [10.99, 20.50, 15.75],
    'Stock': [100, 50, 200]
}
df = pd.DataFrame(data)
print(df)
print()
print(df.head())  # Display the first few rows
print()
filtered_df = df[df['Price'] > 15]
print(filtered_df)
print()
# Add a new column for total value in stock
df['Total Value'] = df['Price'] * df['Stock']
print(df)
print()
#Group by 'Product' and calculate the total stock
grouped_df = df.groupby('Product')['Stock'].sum().reset_index()
print(grouped_df)
```

**Output:-**

```
  Product  Price  Stock
0       A  10.99    100
1       B  20.50     50
2       C  15.75    200

  Product  Price  Stock
0       A  10.99    100
1       B  20.50     50
2       C  15.75    200
```

```
  Product  Price  Stock
1       B     20.50     50
2       C     15.75    200

  Product  Price  Stock  Total Value
0       A    10.99    100       1099.0
1       B    20.50     50       1025.0
2       C    15.75    200       3150.0

  Product  Stock
0       A    100
1       B     50
2       C    200
```

**Source code:-**

```
import pandas as pd

data = {
    'Employee': ['Alice', 'Bob', 'Charlie'],
    'Salary': [70000, 80000, 60000],
    'Department': ['HR', 'Engineering', 'Marketing']
}
df = pd.DataFrame(data)
print(df)
print()

print(df.head())  # Display the first few rows
print()

# Filter employees with a salary greater than 65000
filtered_df = df[df['Salary'] > 65000]
print(filtered_df)
print()

# Add a new column for annual bonus (10% of salary)
df['Bonus'] = df['Salary'] * 0.10
print(df)
print()

# Group by 'Department' and calculate the average salary
grouped_df = df.groupby('Department')['Salary'].mean().reset_index()
print(grouped_df)
```

**Output:-**

```
  Employee   Salary   Department
0    Alice      70000          HR
1      Bob      80000     Engineering
2  Charlie     60000    Marketing

  Employee  Salary   Department
0    Alice     70000          HR
1      Bob     80000    Engineering
2  Charlie     60000    Marketing

  Employee  Salary   Department
0    Alice     70000          HR
1      Bob     80000   Engineering

  Employee  Salary   Department     Bonus
0    Alice      70000          HR          7000.0
1      Bob      80000     Engineering  8000.0
2   Charlie    60000    Marketing     6000.0

    Department   Salary
0  Engineering   80000.0
1          HR       70000.0
2    Marketing   60000.0
```

**Source code:-**

```
import pandas as pd



data = {

    'Date': pd.to_datetime(['2024-01-01', '2024-02-01', '2024-03-01']),

    'Sales': [200, 300, 250],

    'Region': ['North', 'South', 'East']

}

df = pd.DataFrame(data)

print(df)

print()



# Display the first few rows
```

```
print(df.head())

print()

# Filter sales greater than 250

filtered_sales = df[df['Sales'] > 250]

print(filtered_sales)

print()

# Add a new column for sales tax (5%)

df['Sales Tax'] = df['Sales'] * 0.05

print(df)

print()

# Group by 'Region' and sum sales

grouped_sales = df.groupby('Region')['Sales'].sum().reset_index()

print(grouped_sales)
```

**<u>Output:-</u>**

```
        Date     Sales Region
0  2024-01-01    200  North
1  2024-02-01    300  South
2  2024-03-01    250   East


        Date     Sales Region
0  2024-01-01    200  North
1  2024-02-01    300  South
2  2024-03-01    250   East


        Date     Sales Region
1  2024-02-01    300  South
```

```
        Date        Sales  Region   Sales Tax
0  2024-01-01    200   North       10.0
1  2024-02-01    300   South       15.0
2  2024-03-01    250    East        12.5


  Region  Sales
0   East    250
1  North    200
2  South    300
```

**Result:-**The above program has been executed successfully.

## EXPERIMENT-6

**Aim:-** Develop programs for JSON to Python dataframes and Python dataframes to JSON.

**Description:-** Develop a program to convert JSON data to Python DataFrames and vice versa. It supports various JSON formats, ensuring accurate parsing and serialization. The tool offers user-friendly features and customizable output options, facilitating efficient data handling. Ideal for data analysis workflows, it enhances interoperability and streamlines data preparation

**Source code:-**

```
import pandas as pd
import json
def json_to_dataframe(json_data):
    if isinstance(json_data, str):
        json_data = json.loads(json_data)  # Load string to dict if necessary
    df = pd.json_normalize(json_data)  # Normalize semi-structured JSON
    return df
def dataframe_to_json(df, orient='records', indent=4):
    json_data = df.to_json(orient=orient, indent=indent)
    return json_data
# Example usage
if _name___== "_main_":
    # Sample JSON data
    sample_json = '''
    [
        {"name": "Alice", "age": 30, "city": "New York"},
        {"name": "Bob", "age": 25, "city": "Los Angeles"}
    ]
    '''
    # Convert JSON to DataFrame
    df = json_to_dataframe(sample_json)
    print("DataFrame:")
    print(df)
    # Convert DataFrame back to JSON

    json_output = dataframe_to_json(df)

    print("\nJSON Output:")

    print(json_output)
```

**Output:-**

```
  DataFrame:
     name    age        city
0   Alice     30    New York
1    Bob      25    Los Angeles

JSON Output:
[
    {
        "name":"Alice",
        "age":30,
        "city":"New York"
    },
    {
        "name":"Bob",
        "age":25,
        "city":"Los Angeles"
    }
]
```

**Source code:-**

```
def json_to_dataframe(json_data):
    if isinstance(json_data, str):
        json_data = json.loads(json_data)
    df = pd.json_normalize(json_data)
    return df

def dataframe_to_json(df, orient='records', indent=4):
    json_data = df.to_json(orient=orient, indent=indent)
    return json_data
if _name___== "_main_":
    # Sample JSON data
    sample_json = '''
    [
        {"product": "Apples", "price": 1.2, "quantity": 50},
        {"product": "Bananas", "price": 0.5, "quantity": 100},
        {"product": "Cherries", "price": 3.0, "quantity": 20}
     ]
    '''


    # Convert JSON to DataFrame
    df2 = json_to_dataframe(sample_json)
    print("\nDataFrame 2:")
    print(df2)

    # Convert DataFrame back to JSON
    json_output_2 = dataframe_to_json(df2)
    print("\nJSON Output 2:")
    print(json_output_2)
```

**Output:-**

```
  DataFrame 2:
    product    price  quantity
0    Apples     1.2        50
1   Bananas     0.5       100
2  Cherries     3.0        20

JSON Output 2:
[
    {
        "product":"Apples",
        "price":1.2,
        "quantity":50
    },
    {
        "product":"Bananas",
        "price":0.5,
        "quantity":100
    },
    {
        "product":"Cherries",
        "price":3.0,
        "quantity":20
    }
]
```

**Source code:-**

```
def json_to_dataframe(json_data):
   if isinstance(json_data, str):
      json_data = json.loads(json_data)
   df = pd.json_normalize(json_data)
   return df

def dataframe_to_json(df, orient='records', indent=4):
   json_data = df.to_json(orient=orient, indent=indent)
   return json_data
if  _name__==  "_main_":
   sample_json_3 = '''
   [
      {"title": "1984", "author": "Orwell", "published_year": 1949},
      {"title": "To Kill a Mockingbird", "author": "Lee", "published_year": 1960},
      {"title": "The Great Gatsby", "author": "Fitzgerald", "published_year": 1925}
    ]
    '''
   # Convert JSON to DataFrame
   df3 = json_to_dataframe(sample_json_3)
   print("\nDataFrame 3:")
   print(df3)

   # Convert DataFrame back to JSON
   json_output_3 = dataframe_to_json(df3)
   print("\nJSON Output 3:")
   print(json_output_3)
```

**Output:-**

```
DataFrame 3:
                   title      author  published_year
0                   1984      Orwell            1949
1  To Kill a Mockingbird         Lee            1960
2       The Great Gatsby  Fitzgerald            1925

JSON Output 3:
[
   {
      "title":"1984",
      "author":"Orwell",
      "published_year":1949
   },
   {
      "title":"To Kill a Mockingbird",
      "author":"Lee",
        "published_year":1960
    },
    {
        "title":"The Great Gatsby",
        "author":"Fitzgerald",
        "published_year":1925
    }
]
```

**Result:-**The above program has been executed successfully.

EXP.NO: 7 DATE:

## EXPERIMENT-7

**Aim:** Develop a program read the data from any file and apply different types of operations using pandas.

**Description:** Pandas is a powerful and widely-used open-source library in Python designed for data manipulation and analysis. It provides flexible data structures, such as DataFrames and Series, which make it easy to handle structured data.This program will help you learn how to use Pandas to handle and analyze data easily. It will allow you to:

1. **Load Data:**
   - Read data from CSV and Excel files.
   - Handle simple issues like missing values.
2. **Explore Data:**
   - Show basic statistics (like mean and count) of the dataset.
   - Display the first few rows of the data for a quick look.
3. **Clean Data:**
   - Remove duplicate entries.
   - Fill or drop missing values.
4. **Manipulate Data:**
   - Filter data based on specific criteria (e.g., values in a column).
   - Sort data by any column.
5. **Visualize Data:**
   - Create simple plots (like histograms) to understand data distribution.
6. **Save Data:**
   - Export the cleaned or modified data back to a file.

**Source Code:**

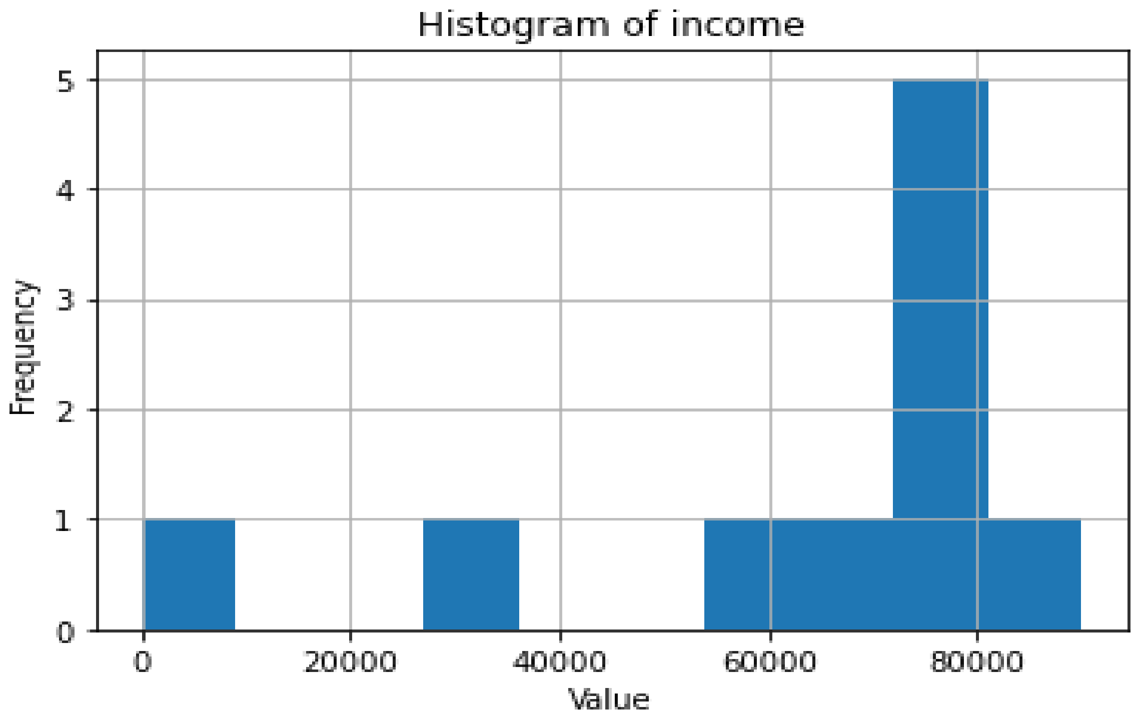
```
import pandas as pd
import matplotlib.pyplot as plt

def load_data(file_path):
    """Load data from a CSV file."""
    data = pd.read_csv(file_path)
    return data

def explore_data(df):
    """Display basic information about the data."""
    print("\nFirst 5 rows:")
    print(df.head())
    print("\nData summary:")
    print(df.describe())

def clean_data(df):
    """Remove duplicates and fill missing values."""
    df = df.drop_duplicates()
    df.fillna(0, inplace=True)  # Fill missing values with 0
    return df

def visualize_data(df):
    """Create a histogram for a specified column."""
    column_name = input("Enter the column name for the histogram: ")
    df[column_name].hist()
    plt.title(f'Histogram of {column_name}')
    plt.xlabel('Value')
    plt.ylabel('Frequency')
    plt.show()

def main():
    # Specify the path to your CSV file here
    file_path = r'C:\Users\nbkr\Desktop\DRAA EXP_7,8,9\example.csv'
  # Change this to your file path if needed

    # Load and explore data
    df = load_data(file_path)
    explore_data(df)

    # Clean data
    df = clean_data(df)

    # Print cleaned data
    print("\nCleaned Data:")
    print(df)

    # Visualize data
    visualize_data(df)

if _name___== "_main_":
    main()
```

## Input:

```
Enter the column name for the histogram: income
```

**Output:**

```
First 5 rows:
   id     name   age  income
0   1  Bhargav  24.0   80000
1   2   Vikram  23.0   80000
2   3  Sainadh  21.0   30000
3   4   Shukur  20.0   69000
4   5   Harsha  24.0   73000

Data summary:
             id        age        income
count  10.00000   9.000000     10.000000
mean    5.50000  26.333333  63700.000000
std     3.02765   6.184658  27765.085989
min     1.00000  20.000000      0.000000
25%     3.25000  23.000000  62250.000000
50%     5.50000  24.000000  74000.000000
75%     7.75000  30.000000  80000.000000
max    10.00000  40.000000  90000.000000

Cleaned Data:
   id      name   age  income
0   1   Bhargav  24.0   80000
1   2    Vikram  23.0   80000
2   3   Sainadh  21.0   30000
3   4    Shukur  20.0   69000
4   5    Harsha  24.0   73000
5   6  Varshith  30.0   75000
6   7      Gina  30.0   80000
7   8       Bob  25.0   60000
8   9    Hannah   0.0       0
9  10       Ian  40.0   90000
```



**Result:** The above program has been successfully executed.

## EXPERIMENT-8

**Aim:** Develop a program for dashboard.

**Description:** This interactive dashboard, built with Python's Dash framework, allows users to visualize data through line graphs. Users can select different categories from a dropdown menu, and the graph updates automatically to display the selected data.

**Key Features:**

1. **Interactive Dropdown:**
   - Choose from three categories: "Category A," "Category B," and "Category C."
2. **Dynamic Graphs:**
   - The graph updates in real-time based on the selected category.
3. **User-Friendly Layout:**
   - Simple design with a title, dropdown, and graph for easy navigation.
4. **Sample Data:**
   - Uses a small set of sample data, which can be easily replaced with real data.

**Technical Details:**

- **Framework:** Built using Dash for web applications.
- **Visualization:** Uses Plotly for interactive graphs.
- **Data Management:** Utilizes Pandas for data handling.

**Usage:** To run the application, ensure Python and the required libraries (Dash, Plotly, Pandas) are installed. Execute the script, and access the dashboard in a web browser at http://127.0.0.1:8050.

This dashboard is a basic template that can be expanded with more features and real-time data integration.

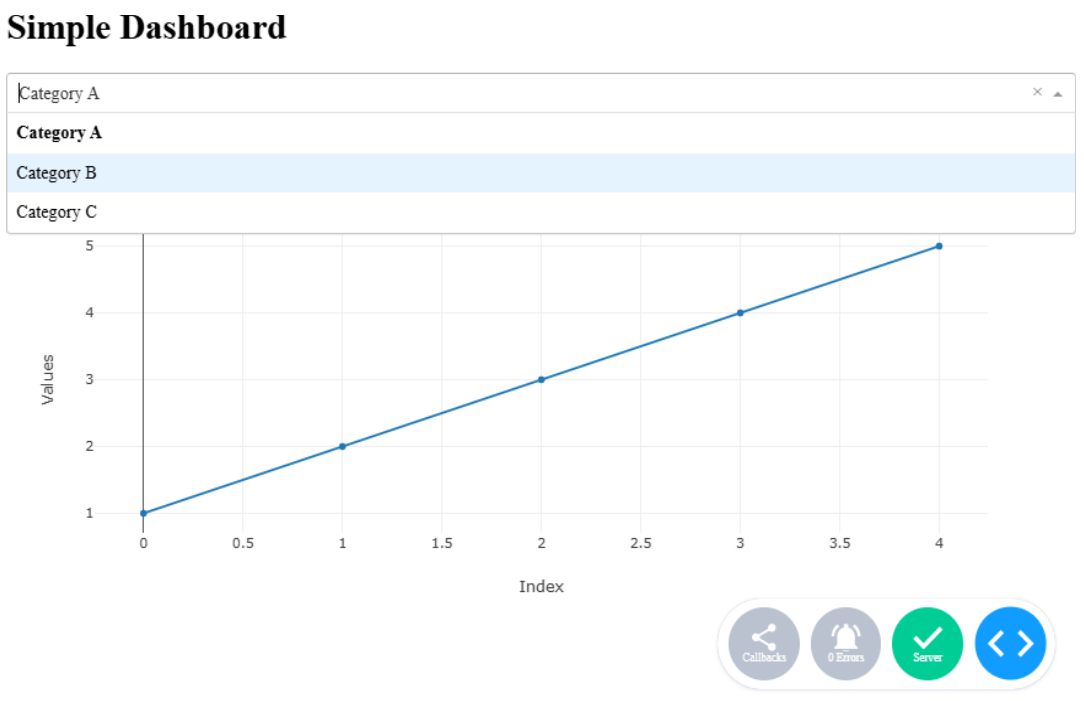
**Source Code:**

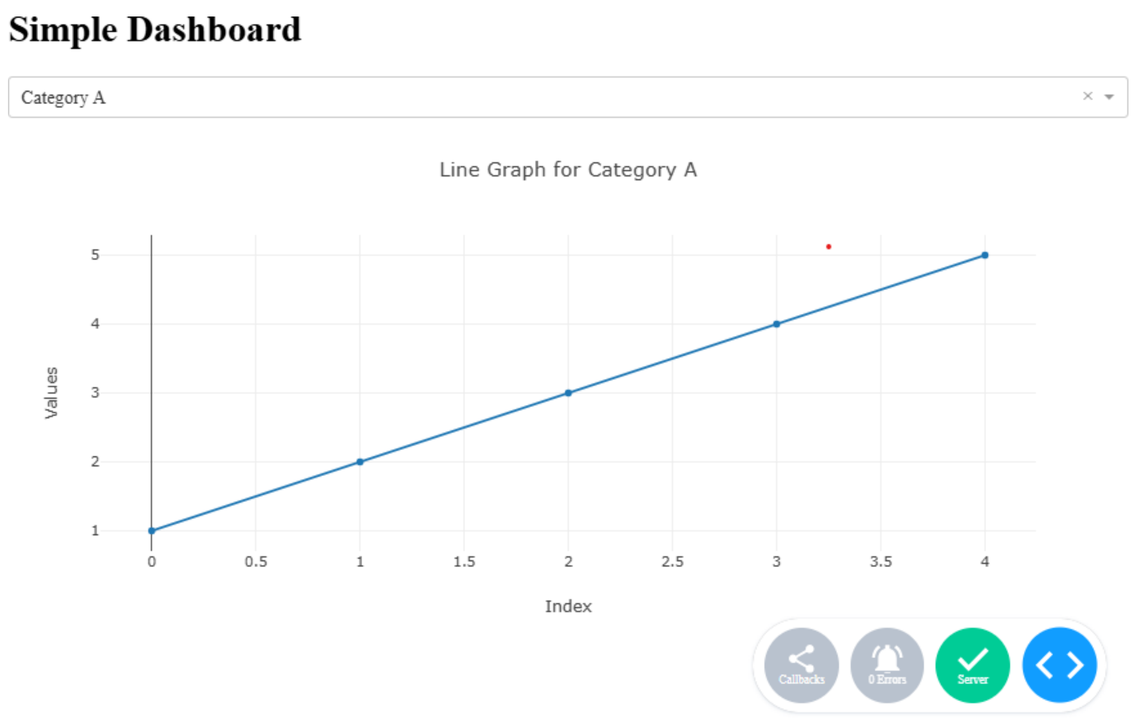
```
#Install dash using below command
pip install dash
#After Installation of dash
import dash
from dash import dcc, html
from dash.dependencies import Input, Output
```

```
import plotly.graph_objs as go
import pandas as pd

# Sample data
data = {
    "Category A": [1, 2, 3, 4, 5],
    "Category B": [5, 4, 3, 2, 1],
    "Category C": [2, 3, 2, 3, 2],
}
df = pd.DataFrame(data)

# Initialize the Dash app
app = dash.Dash(__name__)

# App layout
app.layout = html.Div([
    html.H1("Simple Dashboard"),
    dcc.Dropdown(
        id='category-dropdown',
        options=[
            {'label': 'Category A', 'value': 'Category A'},
            {'label': 'Category B', 'value': 'Category B'},
            {'label': 'Category C', 'value': 'Category C'},
        ],
        value='Category A',
        multi=False,
    ),
    dcc.Graph(id='line-graph'),
])

# Callback to update the graph based on the dropdown selection
@app.callback(
    Output('line-graph', 'figure'),
    Input('category-dropdown', 'value'),
)
```

```
def update_graph(selected_category):
    figure = {
        'data': [
            go.Scatter(
                x=df.index,
                y=df[selected_category],
                mode='lines+markers',
                name=selected_category
            )
        ],
        'layout': go.Layout(
            title=f'Line Graph for {selected_category}',
            xaxis={'title': 'Index'},
            yaxis={'title': 'Values'},
        )
    }
    return figure

# Run the app
if _name___== '_main_':
    app.run_server(debug=True)
```

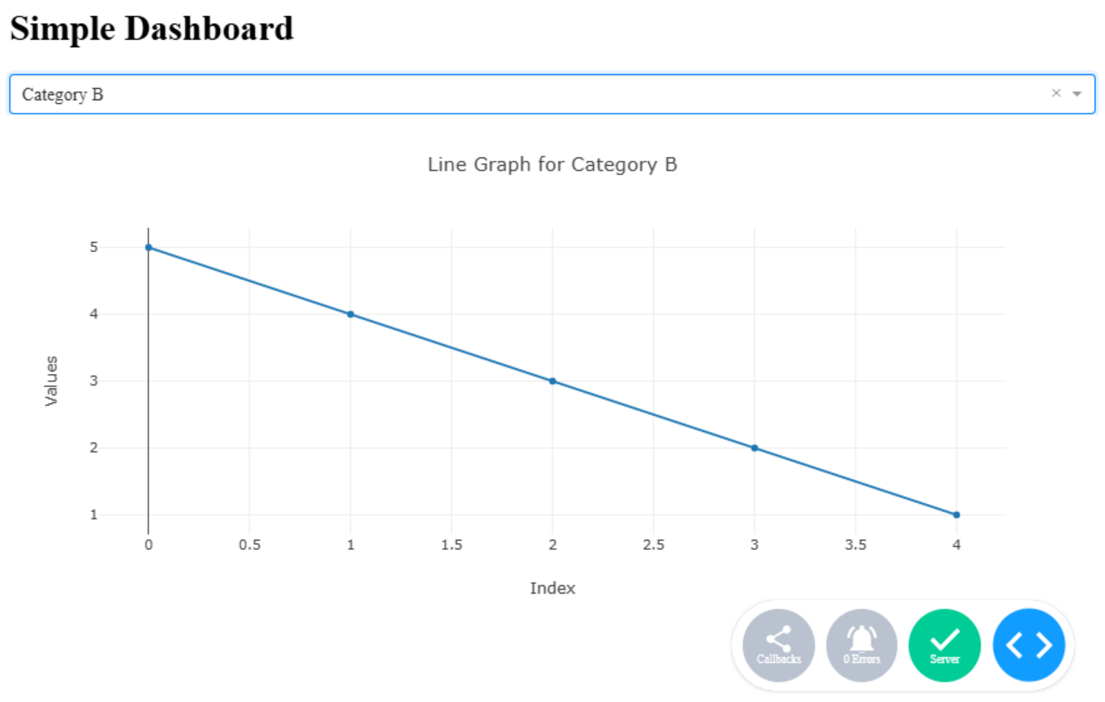
**Output:**





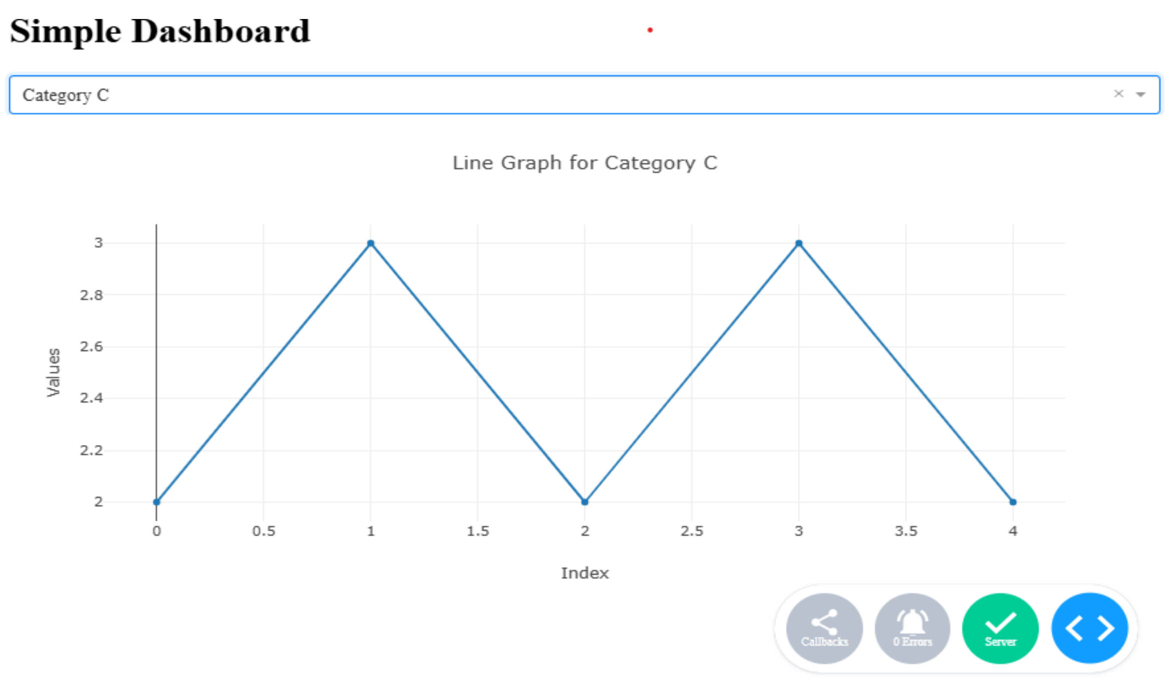
## Simple Dashboard

Category A



## Simple Dashboard

Category B


Simple Dashboard

Category C

**Result:** The above program has been successfully executed.

## EXPERIMENT-9

**Aim:** Develop programs for data visualization using matplotlib.

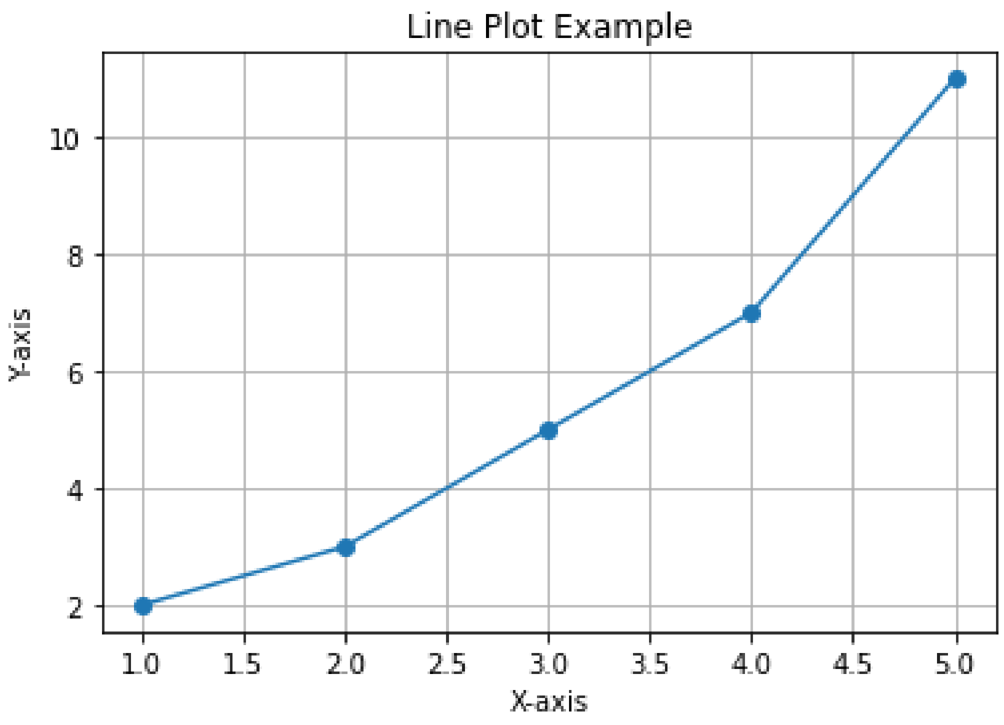
**Description:** Data visualization is the graphical representation of information and data. By using visual elements like charts, graphs, and maps, data visualization tools provide an accessible way to see and understand trends, outliers, and patterns in data.The main goal of using Matplotlib for data visualization is to make complex data easier to understand through visual representations. Here are the key objectives:

1. **Improved Understanding:**
   - Visuals help clarify complex data, making trends and patterns easier to spot.
2. **Effective Communication:**
   - Graphs convey insights quickly, helping stakeholders make informed decisions.
3. **Variety of Visualizations:**
   - Matplotlib offers many plot types (line, bar, pie, scatter, etc.), allowing users to choose the best format for their data.
4. **Customization:**
   - Users can easily customize colors, labels, and titles to enhance clarity and aesthetics.
5. **Integration with Data Analysis:**
   - Works well with libraries like NumPy and Pandas, making it ideal for visualizing analysis results.
6. **Accessibility:**
   - Widely used and well-documented, making it easy for beginners and powerful for advanced users.
7. **Interactivity:**
   - Can create interactive plots, allowing users to explore data dynamically.

**Source Code:**

```
#Line Plot

import matplotlib.pyplot as plt

x = [1, 2, 3, 4, 5]

y = [2, 3, 5, 7, 11]

plt.plot(x, y, marker='o')
plt.title("Line Plot Example")

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.grid()

plt.show()
```

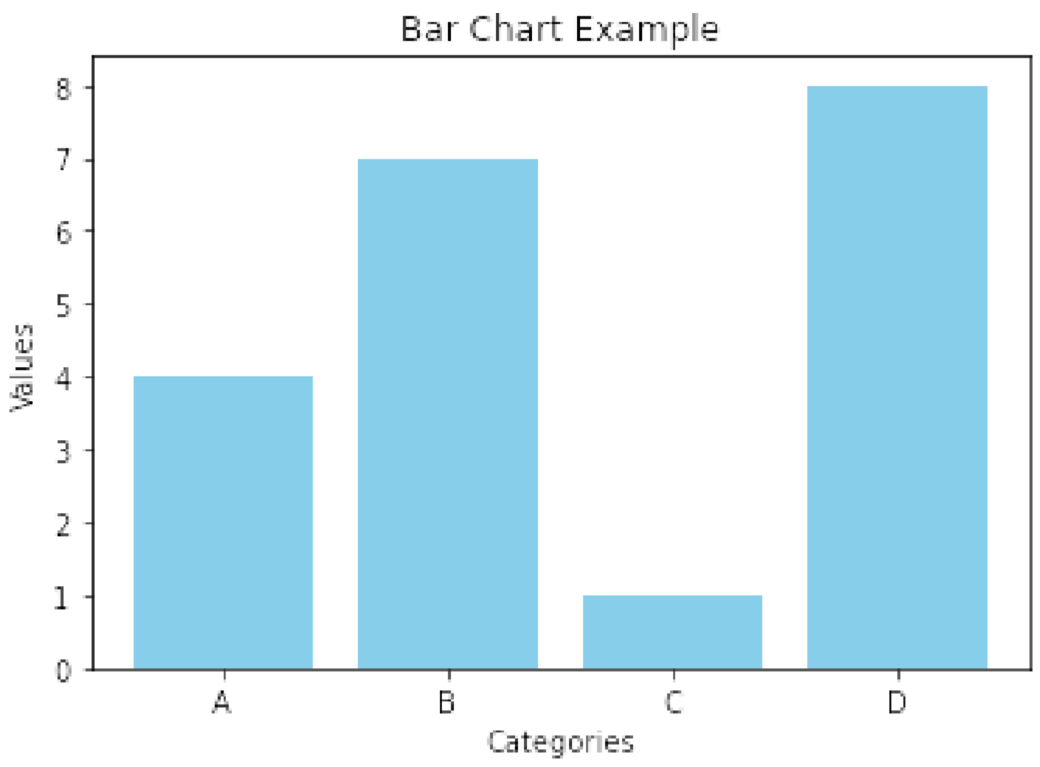
**Output:**



**Source Code:**

```
import matplotlib.pyplot as plt

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

values = [4, 7, 1, 8]

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

plt.title("Bar Chart Example")

plt.xlabel("Categories")

plt.ylabel("Values")

plt.show()
```

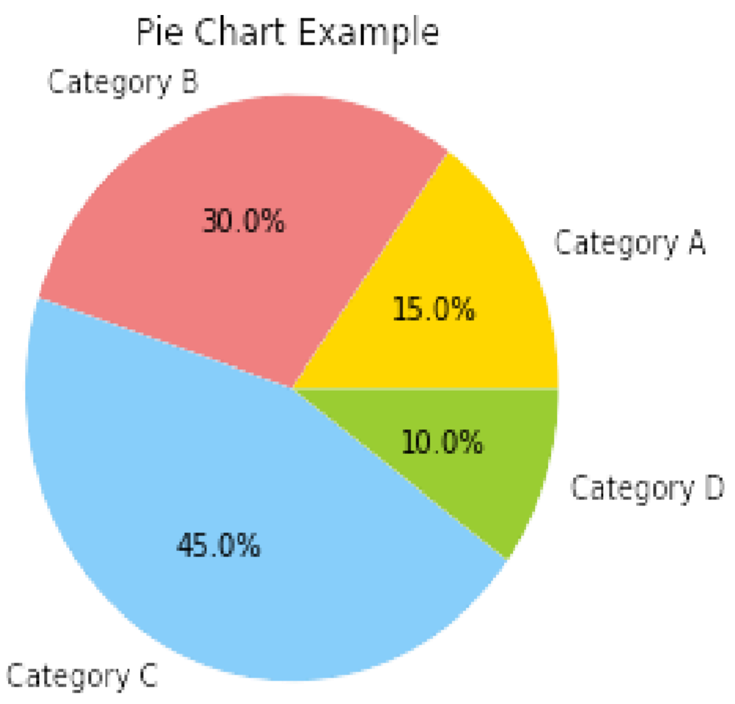
**Output:**



**Source Code:**

```
import matplotlib.pyplot as plt

sizes = [15, 30, 45, 10]

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

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

plt.pie(sizes, labels=labels, colors=colors, autopct='%1.1f%%')

plt.title("Pie Chart Example")

plt.axis('equal')  # Equal aspect ratio ensures that pie chart is circular.

plt.show()
```

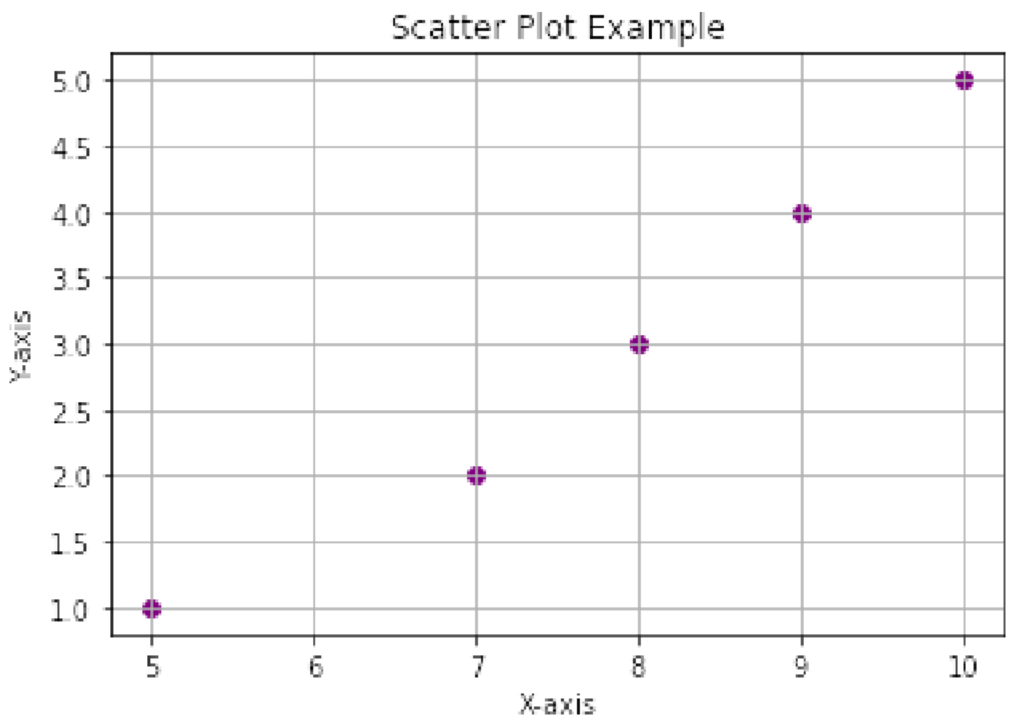
**Output:**



**Source Code:**

```
import matplotlib.pyplot as plt

x = [5, 7, 8, 9, 10]

y = [1, 2, 3, 4, 5]

plt.scatter(x, y, color='purple')

plt.title("Scatter Plot Example")

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.grid()

plt.show()
```

**Output:**



**Result:**The above program has been successfully executed.

# EXPERIMENT-10

**Aim** : Develop a program for Linear Regression model

**Description:** Linear regression is a statistical method used to model the relationship between a dependent variable (often referred to as Y) and one or more independent variables (referred to as X). It assumes a linear relationship between the dependent and independent variables, aiming to fit a straight line through the data points that best explains the observed outcomes.

The equation of a simple linear regression model is:

$$Y = \beta 0 + \beta 1X + \epsilon$$

Where:

- $Y$ is the dependent variable.
- $X$ is the independent variable.
- $\beta 0$ is the intercept (the value of $Y$ when $X = 0$).
- $\beta 1$ is the slope (the change in $Y$ for a one-unit change in $X$).
- $\epsilon$ is the error term (residuals).

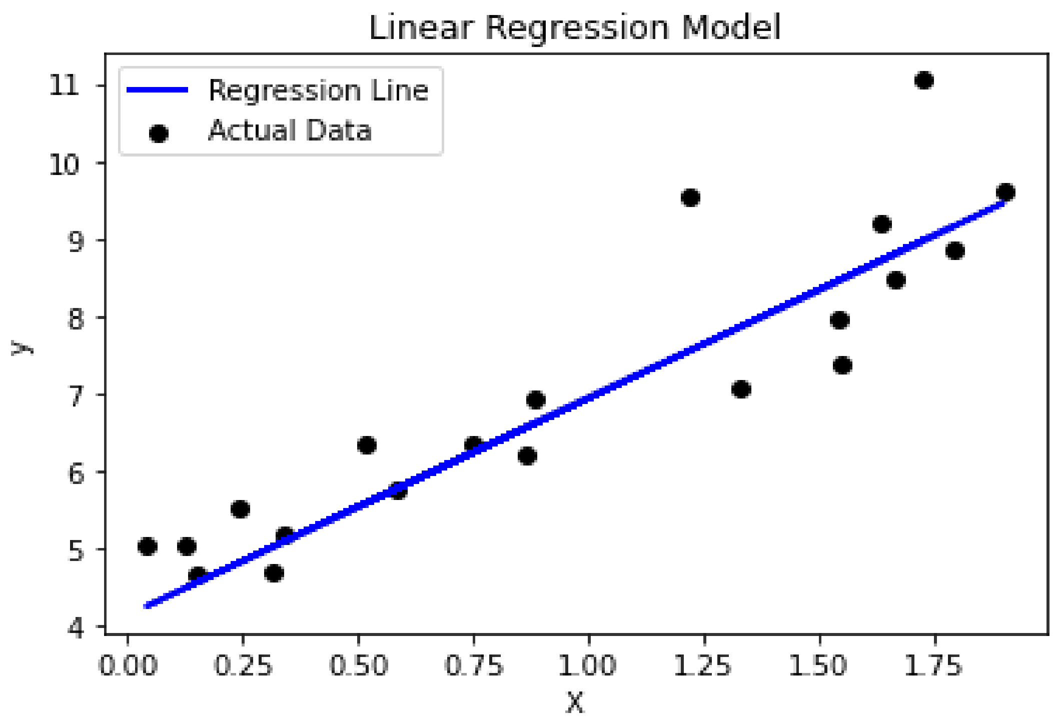
## Source Code:

```
# Import necessary libraries

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error, r2_score

# Generate some example data

np.random.seed(42)

X = 2 * np.random.rand(100, 1)

y = 4 + 3 * X + np.random.randn(100, 1)

# Split the dataset into training and testing sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Create the Linear Regression model

model = LinearRegression()

# Train the model

model.fit(X_train, y_train)

# Make predictions using the testing set

y_pred = model.predict(X_test)

# Print the model coefficients

print("Intercept (β0):", model.intercept_)

print("Slope (β1):", model.coef_)

# Evaluate the model

mse = mean_squared_error(y_test, y_pred)

r2 = r2_score(y_test, y_pred)

print("Mean Squared Error (MSE):", mse)

print("R-squared (R2):", r2)

# Plot the regression line

plt.scatter(X_test, y_test, color='black', label='Actual Data')

plt.plot(X_test, y_pred, color='blue', linewidth=2, label='Regression Line')

plt.xlabel('X')

plt.ylabel('y')

plt.title('Linear Regression Model')

plt.legend()

plt.show()
```

**Output:**

```
Intercept (β0): [4.14291332]
Slope (β1): [[2.79932366]]
Mean Squared Error (MSE): 0.6536995137170021
R-squared (R2): 0.8072059636181392
```



**Result:** The above program have been successfully executed

## EXPERIMENT-11

**Aim:** Develop a program for Decision Tree Regression model.

**Description:** Decision Tree Regression is a non-parametric supervised learning method used for regression tasks. Unlike linear regression, decision tree regression works by splitting the dataset into smaller subsets, eventually leading to leaf nodes where a prediction is made based on the average value of the target variable in that subset. It recursively partitions the data into regions to minimize a cost function, such as mean squared error (MSE).

**Key Advantages:**

- It can model non-linear relationships.
- It is simple to interpret and visualize.
- It does not require feature scaling or normalization.

However, decision trees are prone to overfitting, especially when they are deep, which can lead to poor generalization on unseen data.

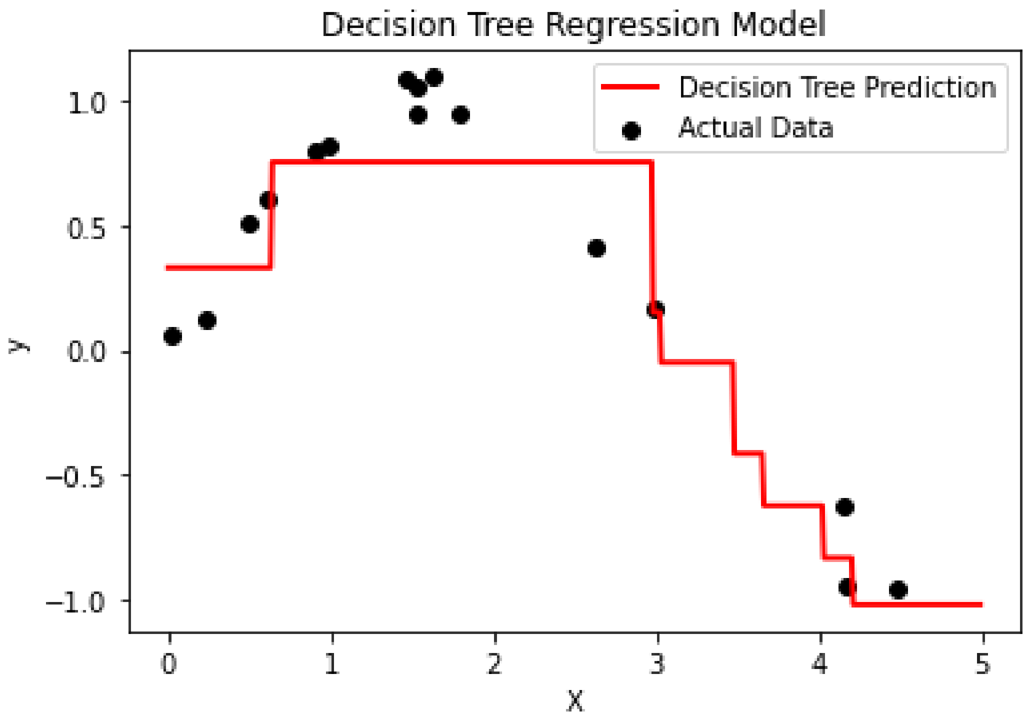
### Source Code:

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
# Generate some example data
np.random.seed(42)
X = np.sort(5 * np.random.rand(80, 1), axis=0)
y = np.sin(X).ravel() + np.random.randn(80) * 0.1  # Adding some noise to the data
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Create a Decision Tree Regressor model
```

```
model = DecisionTreeRegressor(max_depth=3)

# Train the model

model.fit(X_train, y_train)

# Make predictions using the testing set

y_pred = model.predict(X_test)

# Print performance metrics

mse = mean_squared_error(y_test, y_pred)

r2 = r2_score(y_test, y_pred)

print("Mean Squared Error (MSE):", mse)

print("R-squared (R2):", r2)

# Visualize the decision tree regression results

# Plot the actual data and the predicted curve

X_plot = np.arange(0.0, 5.0, 0.01)[:, np.newaxis]

y_plot = model.predict(X_plot)

plt.scatter(X_test, y_test, color='black', label='Actual Data')

plt.plot(X_plot, y_plot, color='red', label='Decision Tree Prediction', linewidth=2)

plt.xlabel('X')

plt.ylabel('y')

plt.title('Decision Tree Regression Model')

plt.legend()

plt.show()
```

**Output:**

```
Mean Squared Error (MSE): 0.04963432432853545


R-squared (R2): 0.8915591153891227
```



**Result:** The above program has been successfully executed

# EXPERIMENT-12

**Aim:** Develop a program for Random Forest Regression model.

**Description:** Random Forest Regression is an ensemble learning method that combines multiple decision trees to improve predictive accuracy and control overfitting. The algorithm builds a large number of decision trees during training and outputs the average prediction of these trees for regression tasks. This reduces the variance observed in individual trees by averaging their predictions, which results in a more robust and generalizable model.

## Key Characteristics:

- **Ensemble Learning**: Combines multiple decision trees (also called "forest") to improve predictive performance.
- **Bagging**: Each tree is trained on a random subset of the data with replacement (bootstrap sampling), reducing overfitting.
- **Feature Randomness**: During the training of each tree, a random subset of features is chosen to split nodes, increasing model diversity.

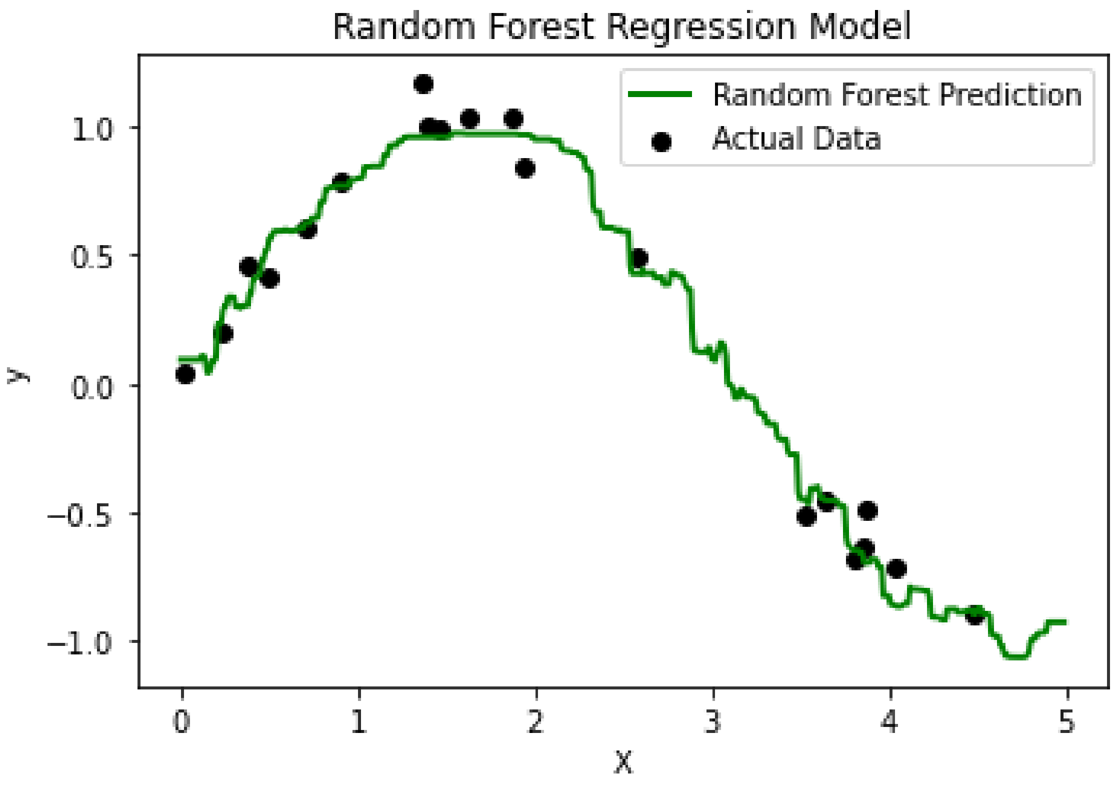
## Source Code:

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
# Generate some example data
np.random.seed(42)
X = np.sort(5 * np.random.rand(100, 1), axis=0)
y = np.sin(X).ravel() + np.random.randn(100) * 0.1  # Adding noise to the sine wave data


# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
# Create a Random Forest Regressor model

# n_estimators: number of trees in the forest

# max_depth: maximum depth of the trees

model = RandomForestRegressor(n_estimators=100, max_depth=5, random_state=42)

# Train the model

model.fit(X_train, y_train)

# Make predictions using the testing set

y_pred = model.predict(X_test)

# Print the model performance metrics

mse = mean_squared_error(y_test, y_pred)

r2 = r2_score(y_test, y_pred)

print("Mean Squared Error (MSE):", mse)

print("R-squared (R2):", r2)

# Visualize the Random Forest regression results

# Plot the actual data and the predicted curve

X_plot = np.arange(0.0, 5.0, 0.01)[:, np.newaxis]

y_plot = model.predict(X_plot)

plt.scatter(X_test, y_test, color='black', label='Actual Data')

plt.plot(X_plot, y_plot, color='green', label='Random Forest Prediction', linewidth=2)

plt.xlabel('X')

plt.ylabel('y')

plt.title('Random Forest Regression Model')

plt.legend()

plt.show()
```

**Output:**

```
Mean Squared Error (MSE): 0.009209907940361771
R-squared (R2): 0.9808016723455534
```



**Result:** The above program has been successfully executed

## EXPERIMENT-13

**Aim:** Develop a model predict the weather forecasting by the customer requirement.

**Description:** The primary goal of this model is to deliver accurate, reliable, and personalized weather forecasts based on specific user requirements. By focusing on the unique needs of various customer segments—such as farmers, event planners, travelers, and outdoor enthusiasts—the model aims to optimize decision-making and enhance overall user experience.

**Data Sources:**
To ensure comprehensive forecasting, the model leverages multiple data inputs, including:

- **Historical Weather Data:** Provides context for understanding long-term climate trends and seasonal patterns.
- **Real-Time Meteorological Observations:** Collects data from weather stations, radar, and satellites for up-to-date accuracy.
- **Satellite Imagery:** Utilizes advanced satellite data to monitor cloud cover, storm developments, and atmospheric changes.
- **Climate Models:** Incorporates large-scale climate models to predict future weather scenarios based on current trends.
- **User-Generated Data:** Collects information from users about their locations and activities to tailor forecasts to their specific needs.

**Key Features:**

1. **Customization:**
   Users can define specific parameters for their forecasts, such as desired temperature ranges, humidity levels, wind conditions, and precipitation likelihood. This feature enables personalized insights, enhancing relevance and utility.
2. **Geolocation Services:**
   The model employs geolocation technology to deliver hyper-local forecasts. By pinpointing exact locations, users receive detailed forecasts that account for microclimates and geographical variations, ensuring accuracy.
3. **Predictive Analytics:**
   Advanced machine learning algorithms analyze historical and real-time data to identify patterns and trends. This predictive capability enhances the accuracy of short-term and long-term forecasts, allowing for better planning and risk management.

4. **Alerts and Notifications:**
   Users can set customized alerts for specific weather conditions that may impact their plans, such as severe thunderstorms, heatwaves, or snowstorms. This proactive approach helps users stay informed and prepared for sudden weather changes.
5. **User-Friendly Interface:**
   An intuitive and interactive interface enables users to easily input their requirements, navigate through forecast options, and visualize data through charts and graphs. This accessibility is essential for a wide range of users, from tech-savvy individuals to those less familiar with technology.
6. **Feedback Loop:**
   Incorporating user feedback is crucial for continuous improvement. The model adapts based on user experiences and preferences, enhancing forecasting accuracy and ensuring the service evolves with changing needs.

**Benefits:**

By delivering tailored weather forecasts, this model empowers users to make informed decisions. Farmers can plan planting and harvesting schedules, event planners can choose optimal dates, and travelers can prepare for conditions at their destinations. Additionally, the model promotes safety by providing timely alerts for severe weather events, helping users mitigate risks.

**Implementation Considerations:**

To successfully implement this model, factors such as data integration, algorithm development, and user engagement strategies must be addressed. Ensuring data accuracy and reliability is critical, as is creating a feedback mechanism that encourages user interaction and satisfaction.

**Conclusion:**

This weather forecasting model ultimately seeks to enhance user experience by providing accurate, relevant, and personalized weather information. By delving into specific customer requirements, it positions itself as a vital tool for planning and decision-making across various sectors.