INDEX

|  |  |  |  |
| --- | --- | --- | --- |
| **S.NO** | **PRACTICALS** | **PAGE NO.** | **SIGN** |
| **1** | 1. Write a python program to print the multiplication table for the given number. 2. Write a python program to check whether the given number isprime or not. 3. Write a python program to find factorial of the given number. | 1-3 |  |
| **2** | Write a python program to implement simple Chatbot. | 4 |  |
| **3** | Write a python program to generate Calendar for the givenmonth and year. | 5 |  |
| **4** | Write a python program to implement Breadth First SearchTraversal. | 6 |  |
| **5** | Write a python program to implement Depth First SearchTraversal. | 7 |  |
| **6** | Write a python program to implement Water Jug Problem. | 8-10 |  |
| **7** | Implement Linear regression using any real data set. | 11-12 |  |
| **8** | Implement Logistic regression using any real data set. | 13-14 |  |
| **9** | Use a real-life data set to implement K-means clustering. | 15-16 |  |
| **10** | Implementing Numpy in lab and use it in: Joining Numpy arrays, Intersection and difference Mean, median, Standard Deviation. | 17 |  |
| **11** | Implementing Pandas Library in lab with available database andsimple commands: Head, Tail. Describe tail, iloc, loc, drop, mean, median, maximum, minimum. | 18-19 |  |
| **12** | Implementing Matplotlib library in lab and use the charts: Barplot, Scatterplot, Pie-chart, Donut chart. | 20-21 |  |
| **13** | Explain the preprocessing of data on any real-life dataset. | 22-23 |  |
| **14** | Write a program to implement the graph coloring problem. | 24 |  |

# PROGRAM 1

## Write a python program to print the multiplication table for the given number.

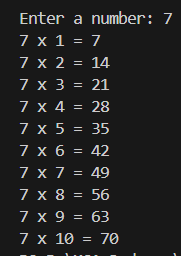
def multiplication\_table(num): for i in range(1, 11):

print(f"{num} x {i} = {num \* i}")

number = int(input("Enter a number: "))

multiplication\_table(number)

**OUTPUT: -**



## Write a python program to check whether the given number is prime or not.

def is\_prime(number): if number <= 1:

return False

for i in range(2, int(number \*\* 0.5) + 1): if number % i == 0:

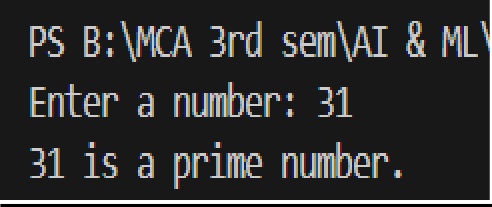
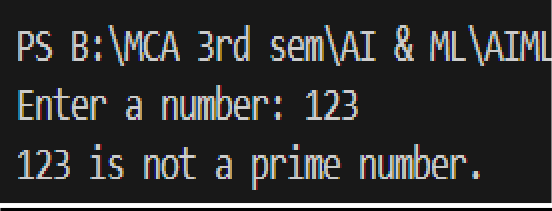
return False return True

num = int(input("Enter a number: ")) if is\_prime(num):

print(f"{num} is a prime number.") else:

print(f"{num} is not a prime number.")

**OUTPUT: -**



## Write a python program to find factorial of the given number.

def factorial(n):

if n == 0 or n == 1: return 1

else:

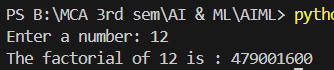
return n \* factorial(n - 1)

# Input from user

num = int(input("Enter a number: ")) result = factorial(num)

print(f"The factorial of {num} is {result}")

**OUTPUT: -**



# PROGRAM 2

## Write a python program to implement simple Chatbot.

def chatbot():

print("Hello! I'm ChatBot. How can I help you today?") while True:

user\_input = input("You: ").lower()

if "hi" in user\_input or "hello" in user\_input: print("ChatBot: Hello! How can I assist you?")

elif "how are you" in user\_input:

print("ChatBot: I'm just a program, but I'm functioning perfectly! How about you?") elif "your name" in user\_input:

print("ChatBot: My name is ChatBot!")

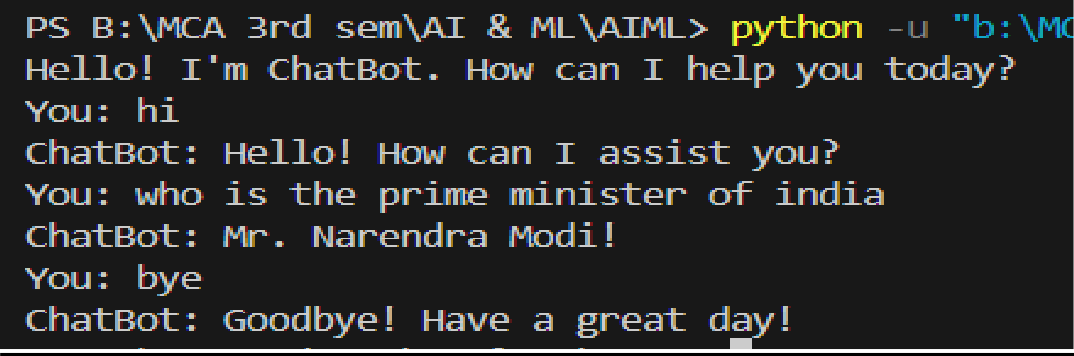
elif "who is the prime minister of india" in user\_input: print("ChatBot: Mr. Narendra Modi!")

elif "bye" in user\_input or "exit" in user\_input: print("ChatBot: Goodbye! Have a great day!") break

else:

print("ChatBot: I'm sorry, I don't understand that. Can you ask something else?") chatbot()

**OUTPUT: -**



# PROGRAM 3

## Write a python program to generate Calendar for the given month and year.

import calendar

def generate\_calendar(year, month):

# Print the calendar for the given month and year print(calendar.month(year, month))

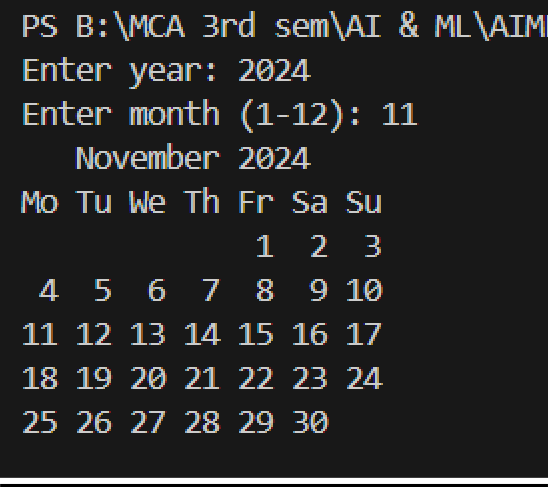
# Input from user

year = int(input("Enter year: "))

month = int(input("Enter month (1-12): "))

generate\_calendar(year, month)

**OUTPUT: -**



# PROGRAM 4

## Write a python program to implement Breadth First Search Traversal.

from collections import deque def bfs(graph, start\_node):

visited = set()

queue = deque([start\_node]) while queue:

node = queue.popleft() if node not in visited:

print(node," -> ", end=" ") visited.add(node)

# Add all unvisited neighbors to the queue for neighbor in graph[node]:

if neighbor not in visited: queue.append(neighbor)

graph = {

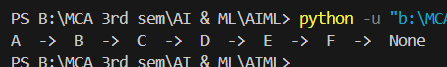
'A': ['B', 'C'], 'B': ['D', 'E'], 'C': ['F'], 'D': [], 'E': ['F'], 'F': []

}

bfs(graph, 'A')

print("None")

**OUTPUT: -**



# PROGRAM 5

## Write a python program to implement Depth First Search Traversal.

def dfs(graph, node, visited): if node not in visited:

print(node, “->” , end=" ") visited.add(node)

for neighbor in graph[node]: dfs(graph, neighbor, visited)

graph = {

'U': ['V', 'W'],

'V': ['X', 'Y'],

'W': ['Z'],

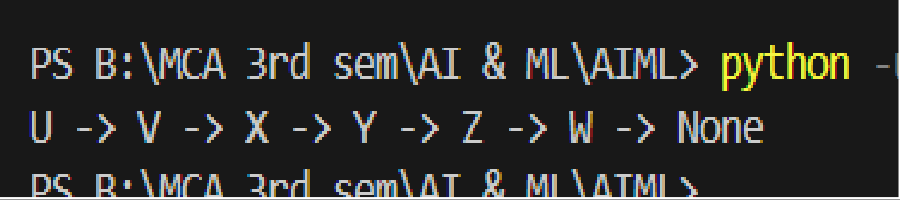
'X': [],

'Y': ['Z'], 'Z': []

}

visited = set() dfs(graph, 'U', visited) print("None")

**OUTPUT: -**



# PROGRAM 6

## Write a python program to implement Water Jug Problem.

from collections import deque

def is\_visited(state, visited\_states): return state in visited\_states

def print\_solution(path): for state in path:

print(f"Jug1: {state[0]} liters, Jug2: {state[1]} liters") print("\nGoal reached!\n")

def water\_jug\_bfs(jug1\_capacity, jug2\_capacity, target): initial\_state = (0, 0)

# Queue to store paths to explore queue = deque([(initial\_state, [])])

visited\_states = set()

while queue:

(jug1, jug2), path = queue.popleft()

# If we have already visited this state, skip it if is\_visited((jug1, jug2), visited\_states):

continue

visited\_states.add((jug1, jug2))

# Add the current state to the path path = path + [(jug1, jug2)]

if jug1 == target or jug2 == target: print\_solution(path)

return True

# All possible operations (transitions) possible\_states = [

(jug1\_capacity, jug2), # Fill Jug1 (jug1, jug2\_capacity), # Fill Jug2 (0, jug2), # Empty Jug1

(jug1, 0), # Empty Jug2

(min(jug1 + jug2, jug1\_capacity), max(0, jug2 - (jug1\_capacity - jug1))), # Pour Jug2 into Jug1

(max(0, jug1 - (jug2\_capacity - jug2)), min(jug1 + jug2, jug2\_capacity)) # Pour Jug1 into Jug2

]

# Enqueue all possible states if they haven't been visited for state in possible\_states:

if not is\_visited(state, visited\_states): queue.append((state, path))

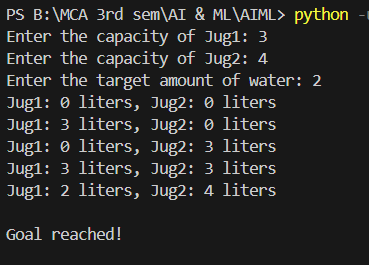
print("No solution found.") return False

jug1\_capacity = int(input("Enter the capacity of Jug1: "))

jug2\_capacity = int(input("Enter the capacity of Jug2: ")) target = int(input("Enter the target amount of water: "))

water\_jug\_bfs(jug1\_capacity, jug2\_capacity, target)

**OUTPUT: -**



# PROGRAM 7

## Implement Linear regression using any real data set.

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.datasets import fetch\_california\_housing from sklearn.metrics import mean\_squared\_error

# Load the California Housing dataset california = fetch\_california\_housing()

X = california.data y = california.target

# Split the data 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 Linear Regression model model = LinearRegression()

# Train the model model.fit(X\_train, y\_train)

# Make predictions

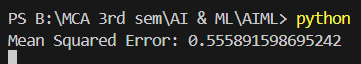
y\_pred = model.predict(X\_test)

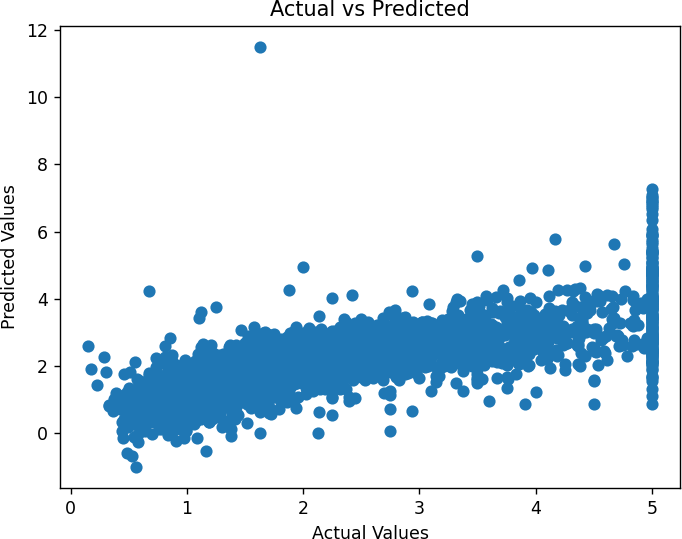
# Print the Mean Squared Error

print("Mean Squared Error:", mean\_squared\_error(y\_test, y\_pred))

# Plotting actual vs predicted values plt.scatter(y\_test, y\_pred) plt.xlabel("Actual Values") plt.ylabel("Predicted Values") plt.title("Actual vs Predicted") plt.show()

**OUTPUT: -**





**PROGRAM 8**

## Implement Logistic regression using any real data set.

import numpy as np

from sklearn.model\_selection import train\_test\_split from sklearn.linear\_model import LogisticRegression from sklearn import datasets

from sklearn.metrics import (confusion\_matrix, precision\_score, recall\_score, f1\_score, accuracy\_score)

import matplotlib.pyplot as plt iris = datasets.load\_iris()

X = iris.data y = iris.target

# We will classify only two classes for binary classification X = X[y != 2]

y = y[y != 2]

# Split the dataset

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42) # Create a Logistic Regression model

log\_reg = LogisticRegression()

# Train the model log\_reg.fit(X\_train, y\_train) y\_pred = log\_reg.predict(X\_test)

y\_prob = log\_reg.predict\_proba(X\_test)[:, 1] # Probability estimates for ROC

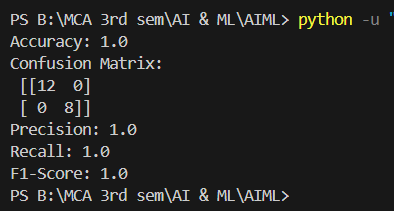
accuracy = accuracy\_score(y\_test, y\_pred) print("Accuracy:", accuracy)

conf\_matrix = confusion\_matrix(y\_test, y\_pred) print("Confusion Matrix:\n", conf\_matrix) precision = precision\_score(y\_test, y\_pred) print("Precision:", precision)

recall = recall\_score(y\_test, y\_pred) print("Recall:", recall)

f1 = f1\_score(y\_test, y\_pred) print("F1-Score:", f1)

**OUTPUT: -**



# PROGRAM 9

## Use a real-life data set to implement K-means clustering.

import numpy as np

from sklearn.datasets import load\_iris from sklearn.cluster import KMeans import matplotlib.pyplot as plt

# Load the Iris dataset iris = load\_iris()

X = iris.data

# Create the KMeans model kmeans = KMeans(n\_clusters=3)

# Fit the model kmeans.fit(X)

# Get the cluster centroids

centroids = kmeans.cluster\_centers\_ labels = kmeans.labels\_

# Plot the clusters

plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='rainbow')

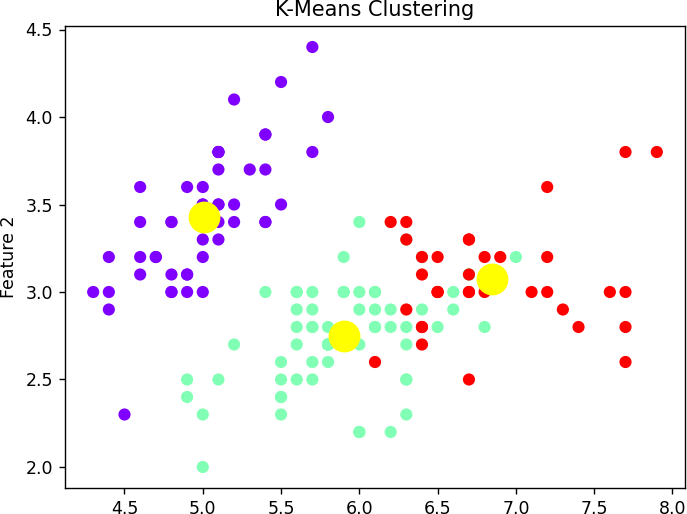
plt.scatter(centroids[:, 0], centroids[:, 1], s=300, c='yellow')

plt.xlabel("Feature 1")

plt.ylabel("Feature 2") plt.title("K-Means Clustering")

plt.show()

**OUTPUT: -**



# PROGRAM 10

## Implementing Numpy in lab and use it in: Joining Numpy arrays , Intersection & Difference , Mean , median , Standard Deviation.

import numpy as np

# Joining Numpy arrays

array1 = np.array([11, 22, 33, 44])

array2 = np.array([45, 56, 67,78])

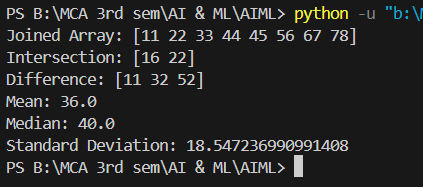
joined\_array = np.concatenate((array1, array2)) print("Joined Array:", joined\_array)

# Intersection and Difference

array3 = np.array([11, 22, 32, 16, 52])

array4 = np.array([41, 22, 16, 71]) intersection = np.intersect1d(array3, array4) difference = np.setdiff1d(array3, array4) print("Intersection:", intersection) print("Difference:", difference)

# Mean, Median, Standard Deviation data = np.array([10, 20, 60, 40, 50]) print("Mean:", np.mean(data)) print("Median:", np.median(data)) print("Standard Deviation:", np.std(data)) **OUTPUT: -**



# PROGRAM 11

## Implementing Pandas Library in lab with available data base and simple commands: Head, Tail. Describe, tail, iloc, loc, drop, mean, median, maximum, minimum.

import pandas as pd

# Create a sample DataFrame data = {'A': [1, 2, 3, 4, 5],

'B': [10, 20, 30, 40, 50],

'C': [100, 200, 300, 400, 500]}

df = pd.DataFrame(data)

# Basic Pandas Commands print(df.head()) # First 5 rows print(df.tail()) # Last 5 rows print(df.describe()) # Summary of data

# Using iloc and loc

print(df.iloc[0]) # First row using iloc print(df.loc[0]) # First row using loc

# Dropping a column

df\_dropped = df.drop('B', axis=1) print(df\_dropped)

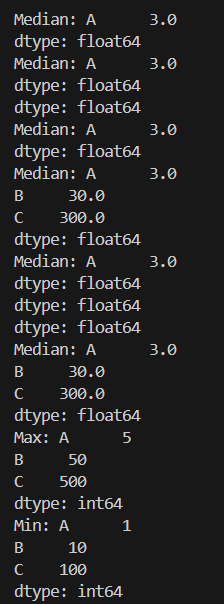
# Mean, Median, Max, Min print("Mean:", df.mean())

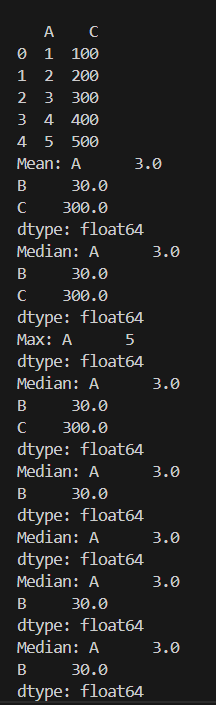
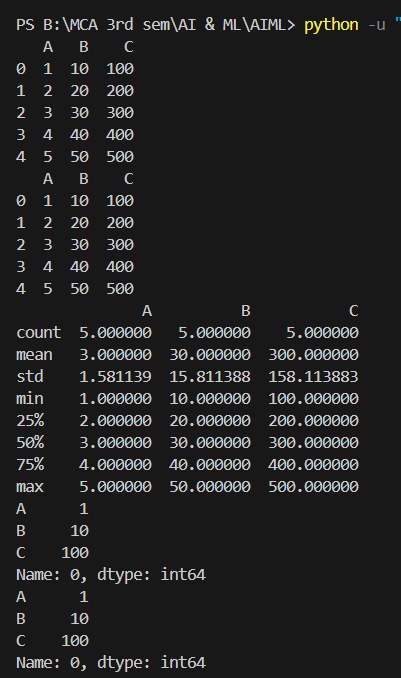
print("Median:", df.median())

print("Max:", df.max())

print("Min:", df.min())

**OUTPUT: -**





# PROGRAM 12

## Implementing Matplotlib library in lab and use the charts: Bar plot, Scatter plot, Pie-chart, Donut- chart.

import matplotlib.pyplot as plt

# Bar Plot

categories = ['A', 'B', 'C'] values = [15, 10, 25] plt.bar(categories, values) plt.title("Bar Plot") plt.show()

# Scatter Plot

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

y = [2, 4, 6, 8, 10]

plt.scatter(x, y) plt.title("Scatter Plot") plt.show()

# Pie Chart

sizes = [10, 30, 45, 20] labels = ['A', 'B', 'C', 'D']

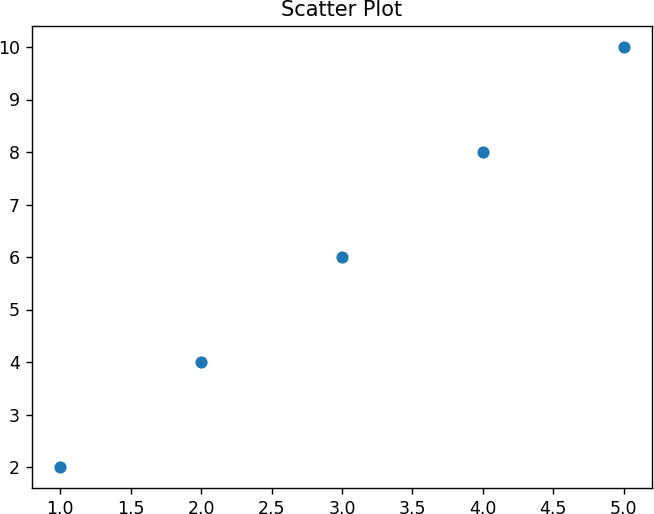
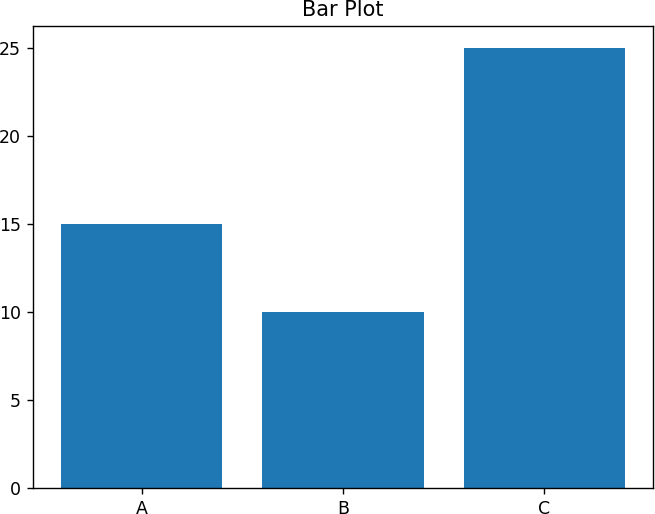
plt.pie(sizes, labels=labels) plt.title("Pie Chart") plt.show()

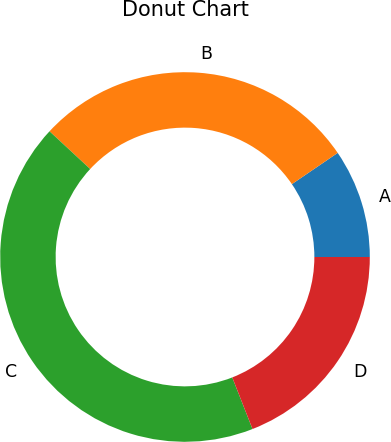
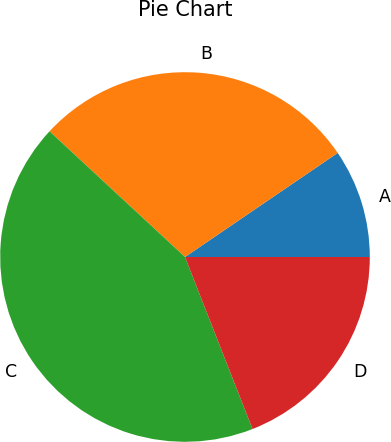
# Donut Chart

plt.pie(sizes, labels=labels, wedgeprops=dict(width=0.3)) plt.title("Donut Chart")

plt.show()

**OUTPUT: -**





# PROGRAM 13

## Explain preprocessing of data on any real-life dataset.

import pandas as pd

from sklearn.datasets import load\_iris

from sklearn.preprocessing import StandardScaler

# Load the Iris dataset iris = load\_iris()

# Convert to a pandas DataFrame

data = pd.DataFrame(iris.data, columns=iris.feature\_names)

# Add the target column to the DataFrame data['species'] = iris.target

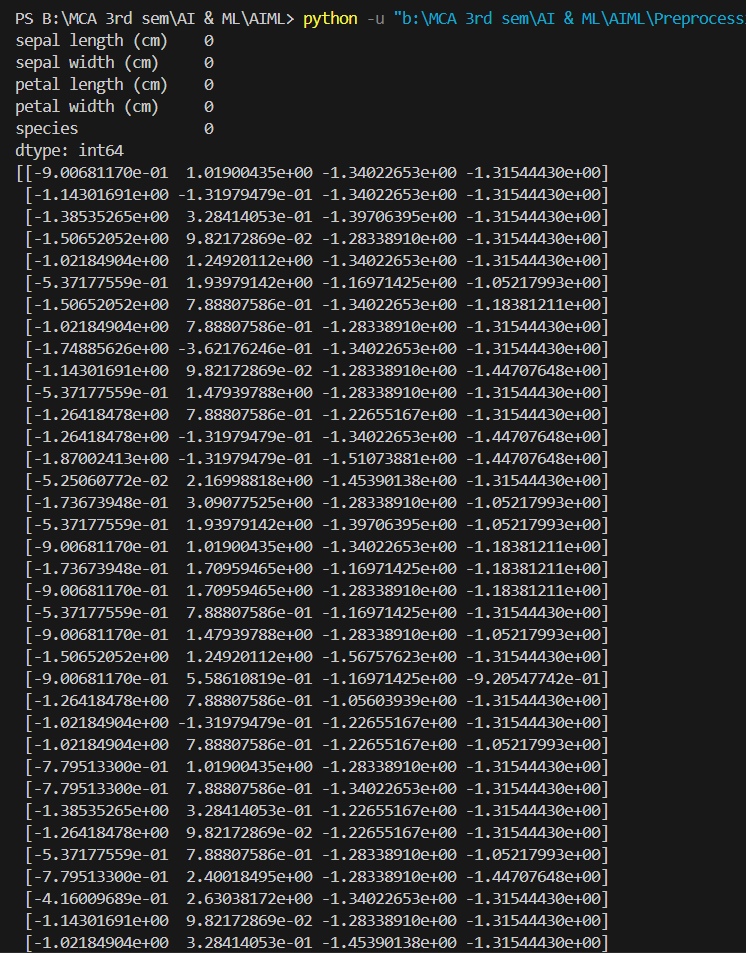
# Checking for missing values print(data.isnull().sum())

# Standardizing the data (excluding the target column) scaler = StandardScaler()

scaled\_data = scaler.fit\_transform(data.iloc[:, :-1])

# Print the standardized features print(scaled\_data)

**OUTPUT: -**



# PROGRAM 14

## Write a program to implement the graph coloring problem.

def graph\_coloring(graph, num\_colors): color = [-1] \* len(graph)

# Assign the first color to the first node color[0] = 0

for node in range(1, len(graph)):

# Find colors that are assigned to the neighboring nodes available\_colors = [True] \* num\_colors

for neighbor in graph[node]: if color[neighbor] != -1:

available\_colors[color[neighbor]] = False # Assign the first available color

for clr in range(num\_colors): if available\_colors[clr]:

color[node] = clr break

return color

# Example graph as adjacency list

graph = { 0: [1, 2], 1: [0, 2], 2: [0, 1, 3], 3: [2]}

result = graph\_coloring(graph, 3) print("Assigned Colors:", result)

**OUTPUT: -**

