**SURYADATTA COLLEGE OF MANAGEMENT**

**INFORMATION RESEARCH & TECHNOLOGY**

**BAVDHAN, PUNE - 411021**

**CS-505-MJP: Lab Course on CS-502-MJ**

**(Artificial Intelligence)**

**Submitted by**

* **NAME OF STUDENTS ROLL NO**

1. **Omkar Joshi 2023100309**

**Under the Guidance of**

**Mr. DEEPAK SINGH**

**SUBMITTED IN PARTIAL FULLFILLMENT OF MASTER OF SCIENCE (COMPUTER SCIENCE)**

**SEM-I**

**SAVITRIBAI PHULE PUNE UNIVERSITY**

**For Academic Year 2023-2024**

**1) Practical on basic programs using python for introducing and using python**

**environment such as,**

**a) Program to print multiplication table for given no.**

**b) Program to check whether the given no is prime or not.**

**c) Program to find factorial of the given no and similar program**

1. **Code::**

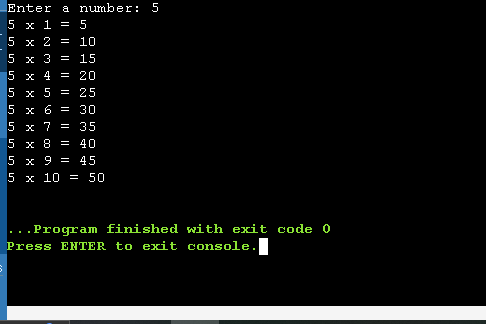
**number = int(input("Enter a number: "))**

**# Print the multiplication table.**

**for i in range(1, 11):**

**print(number, "x", i, "=", number \* i)**

**output:**

****

**b)code::**

**# Python program to check if the given no is prime or not**

**# take input from the user**

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

**# check if the number is prime**

**for i in range(2, num):**

**if (num % i) == 0:**

**print(num, "is not a prime number")**

**break**

**else:**

**print(num, "is a prime number")**

**Output::**

**A screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generated**

**c)code::**

**# Get user input for the number**

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

**# Check if the number is non-negative**

**if num < 0:**

**print("Factorial is not defined for negative numbers.")**

**else:**

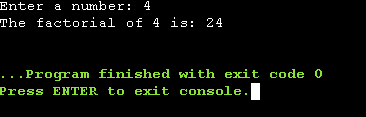
**result = 1**

**for i in range(1, num + 1):**

**result \*= i**

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

**Output:**

****

**Q2)** **Write a program to implement List Operations Nested list, Length, Concatenation, Membership ,Iteration ,Indexing and Slicing List Methods Add, Extend & Delete.**

**Code:**

**nested\_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]**

**# Length of the list**

**length\_of\_list = len(nested\_list)**

**print("Length of the list:", length\_of\_list)**

**# Concatenation of two lists**

**list1 = [1, 2, 3]**

**list2 = [4, 5, 6]**

**concatenated\_list = list1 + list2**

**print("Concatenated list:", concatenated\_list)**

**# Membership test**

**element\_to\_check = 2**

**membership\_result = element\_to\_check in list1**

**print(f"Is {element\_to\_check} present in the list? {membership\_result}")**

**# Iterating through the list**

**print("Iterating through the nested list:")**

**for sublist in nested\_list:**

**for element in sublist:**

**print(element, end=' ')**

**print()**

**# Indexing**

**index\_to\_find = 5**

**index\_result = concatenated\_list.index(index\_to\_find)**

**print(f"Index of {index\_to\_find} in the concatenated list: {index\_result}")**

**# Slicing**

**sliced\_list = concatenated\_list[1:4]**

**print("Sliced list:", sliced\_list)**

**# List method - Append**

**list\_to\_append = [7, 8, 9]**

**list1.append(list\_to\_append)**

**print("List after append:", list1)**

**# List method - Extend**

**list1.extend(list2)**

**print("List after extend:", list1)**

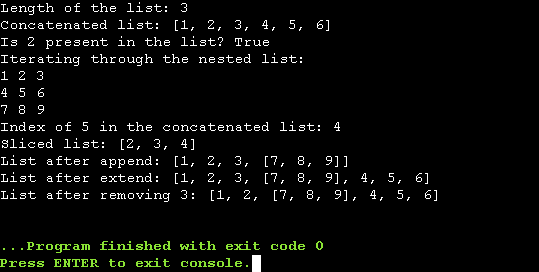
**# List method - Remove**

**element\_to\_remove = 3**

**list1.remove(element\_to\_remove)**

**print(f"List after removing {element\_to\_remove}:", list1)**

**output::**

****

**Q3)Write a program to Illustrate Different Set Operations.**

**Code:;**

**# Creating two sets**

**set1 = {1, 2, 3, 4, 5}**

**set2 = {3, 4, 5, 6, 7}**

**# Union of two sets**

**union\_set = set1.union(set2)**

**print("Union of set1 and set2:", union\_set)**

**# Intersection of two sets**

**intersection\_set = set1.intersection(set2)**

**print("Intersection of set1 and set2:", intersection\_set)**

**# Difference between two sets**

**difference\_set = set1.difference(set2)**

**print("Difference between set1 and set2:", difference\_set)**

**# Symmetric Difference between two sets**

**symmetric\_difference\_set = set1.symmetric\_difference(set2)**

**print("Symmetric Difference between set1 and set2:", symmetric\_difference\_set)**

**# Checking for subset**

**is\_subset = set1.issubset(set2)**

**print("Is set1 a subset of set2?", is\_subset)**

**# Checking for superset**

**is\_superset = set1.issuperset(set2)**

**print("Is set1 a superset of set2?", is\_superset)**

**# Adding an element to the set**

**set1.add(6)**

**print("Set1 after adding element 6:", set1)**

**# Removing an element from the set**

**set2.remove(7)**

**print("Set2 after removing element 7:", set2)**

**Output::A screenshot of a computer

Description automatically generated**

**Q4) Write a program to implement Simple Chatbot.**

**Import random**

**# Define a list of predefined responses**

**responses = {**

**"hello": ["Hi there!", "Hello!", "Hey!"],**

**"how are you": ["I'm just a computer program, but I'm doing well. How can I assist you?", "I'm good. What can I do for you?"],**

**"bye": ["Goodbye!", "See you later!", "Bye now!"],**

**"default": ["I'm not sure I understand.", "Could you please rephrase that?", "I don't have an answer for that."]**

**}**

**# Function to generate a response**

**def chatbot\_response (user\_input):**

**user\_input = user\_input.lower()**

**for key in responses:**

**if key in user\_input:**

**return random.choice(responses[key])**

**return random.choice(responses["default"])**

**# Main loop to run the chatbot**

**print("Chatbot: Hi! How can I assist you today?")**

**while True:**

**user\_input = input("You: ")**

**if user\_input.lower() == "bye":**

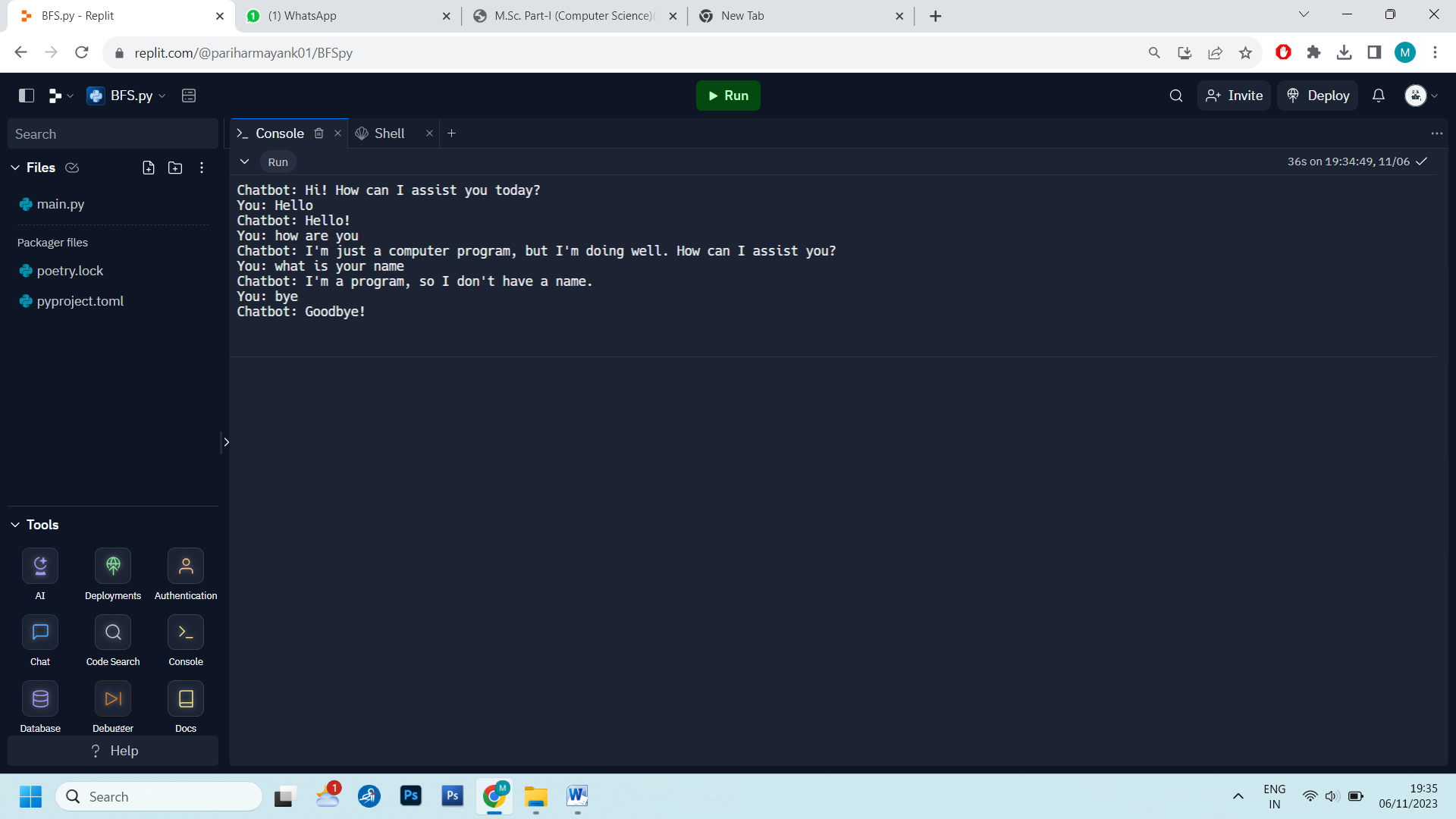
**print("Chatbot: Goodbye!")**

**break**

**response = chatbot\_response(user\_input)**

**print("Chatbot:", response)**

**Output:-**



**Q5) Write a program to implement Breadth First Search Traversal**

**from collections import defaultdict, deque**

**class Graph:**

**def \_init\_(self):**

**self.graph = defaultdict(list)**

**def add\_edge(self, u, v):**

**self.graph[u].append(v)**

**def bfs(self, start):**

**visited = set()**

**queue = deque([start])**

**result = []**

**while queue:**

**vertex = queue.popleft()**

**if vertex not in visited:**

**visited.add(vertex)**

**result.append(vertex)**

**for neighbor in self.graph[vertex]:**

**if neighbor not in visited:**

**queue.append(neighbor)**

**return result**

**# Example usage:**

**if \_\_name\_\_ == "\_main\_":**

**g = Graph()**

**g.add\_edge(0, 1)**

**g.add\_edge(0, 2)**

**g.add\_edge(1, 2)**

**g.add\_edge(2, 0)**

**g.add\_edge(2, 3)**

**g.add\_edge(3, 3)**

**start\_vertex = 2**

**bfs\_result = g.bfs(start\_vertex)**

**print(f"Breadth-First Traversal starting from vertex {start\_vertex}:")**

**print(bfs\_result)**

**Output:-**

**Q6) Write a program to implement Depth First Search Traversal.**

**from collections import defaultdict**

**class Graph:**

**def \_init\_(self):**

**self.graph = defaultdict(list)**

**def add\_edge(self, u, v):**

**self.graph[u].append(v)**

**def dfs\_util(self, vertex, visited):**

**visited.add(vertex)**

**print(vertex, end=' ')**

**for neighbor in self.graph[vertex]:**

**if neighbor not in visited:**

**self.dfs\_util(neighbor, visited)**

**def dfs(self, start):**

**visited = set()**

**print("Depth-First Traversal:")**

**self.dfs\_util(start, visited)**

**# Example usage:**

**if \_\_name\_\_ == "\_main\_":**

**g = Graph()**

**g.add\_edge (0, 1)**

**g.add\_edge(0, 2)**

**g.add\_edge(1, 2)**

**g.add\_edge(2, 0)**

**g.add\_edge(2, 3)**

**g.add\_edge(3, 3)**

**start\_vertex = 2**

**gods(start\_vertex)**

**Output:-**

****

**Q7) Write a program to implement Water Jug Problem**

**def water\_jug\_problem(jug1\_capacity, jug2\_capacity, target):**

**jug1 = 0**

**jug2 = 0**

**steps = []**

**def is\_goal\_state(jug1, jug2, target):**

**return jug1 == target or jug2 == target**

**def fill\_jug1():**

**nonlocal jug1**

**jug1 = jug1\_capacity**

**steps.append(f"Fill jug1 ({jug1}L)")**

**def fill\_jug2():**

**nonlocal jug2**

**jug2 = jug2\_capacity**

**steps.append(f"Fill jug2 ({jug2}L)")**

**def empty\_jug1():**

**nonlocal jug1**

**jug1 = 0**

**steps.append(f"Empty jug1 (0L)")**

**def empty\_jug2():**

**nonlocal jug2**

**jug2 = 0**

**steps.append(f"Empty jug2 (0L)")**

**def pour\_jug1\_to\_jug2():**

**nonlocal jug1, jug2**

**space\_in\_jug2 = jug2\_capacity - jug2**

**if jug1 <= space\_in\_jug2:**

**jug2 += jug1**

**jug1 = 0**

**else:**

**jug1 -= space\_in\_jug2**

**jug2 = jug2\_capacity**

**steps.append(f"Pour from jug1 to jug2 ({jug1}L -> {jug2}L)")**

**def pour\_jug2\_to\_jug1():**

**nonlocal jug1, jug2**

**space\_in\_jug1 = jug1\_capacity - jug1**

**if jug2 <= space\_in\_jug1:**

**jug1 += jug2**

**jug2 = 0**

**else:**

**jug2 -= space\_in\_jug1**

**jug1 = jug1\_capacity**

**steps.append(f"Pour from jug2 to jug1 ({jug2}L -> {jug1}L)")**

**while not is\_goal\_state(jug1, jug2, target):**

**if jug1 == 0:**

**fill\_jug1()**

**elif jug1 > 0 and jug2 < jug2\_capacity:**

**pour\_jug1\_to\_jug2()**

**elif jug1 > 0:**

**empty\_jug2()**

**steps.append(f"Goal reached: {target}L in jug1")**

**for step in steps:**

**print(step)**

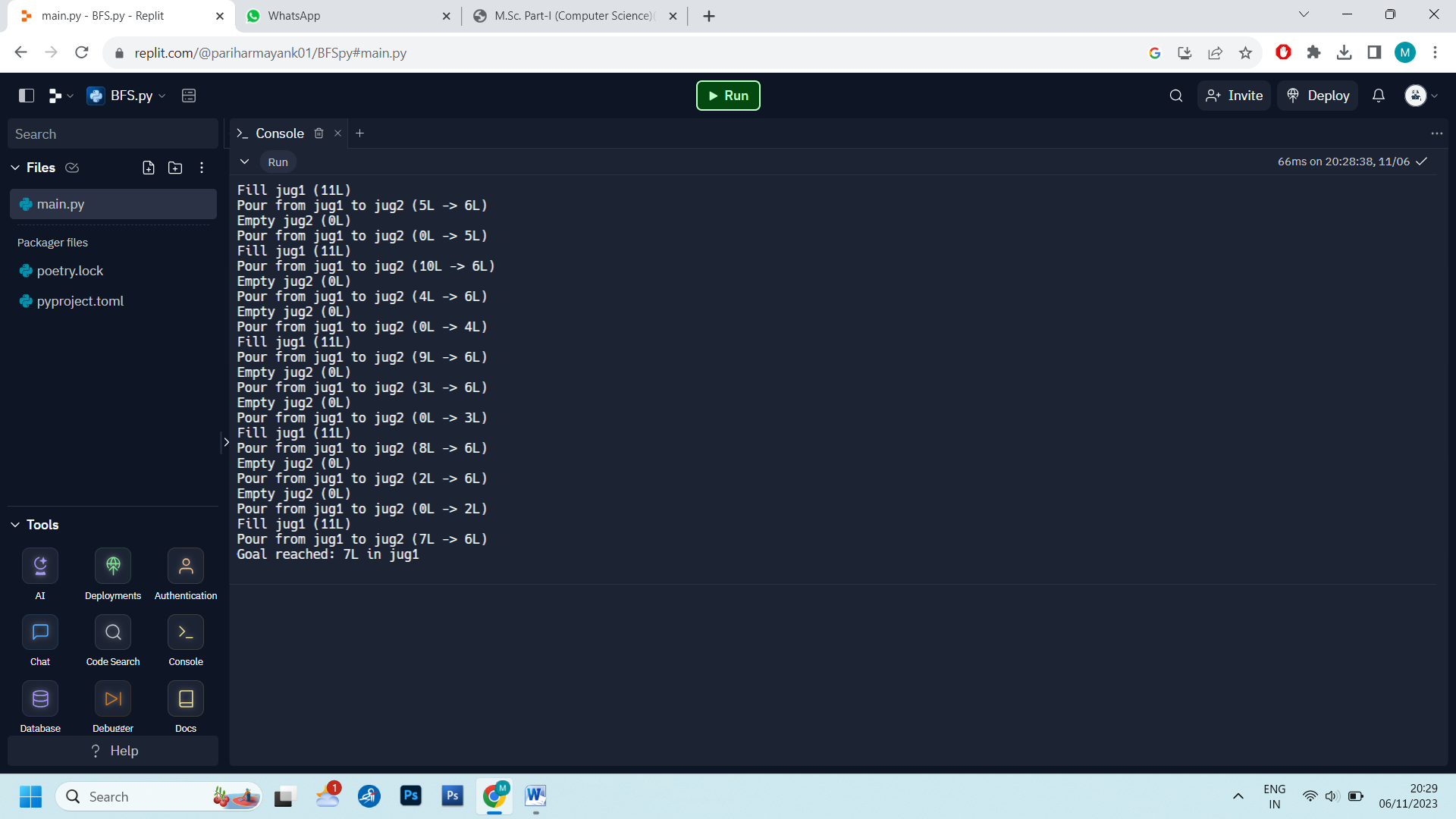
**# Example usage:**

**jug1\_capacity = 11**

**jug2\_capacity = 6**

**target = 7**

**water\_jug\_problem (jug1\_capacity, jug2\_capacity, target)**

**Output:-** 

**8)Write a program to implement K -Nearest Neighbor algorithm**.

**import plotly.graph\_objects as go**

**import numpy as np**

**from sklearn.datasets import make\_moons**

**from sklearn.model\_selection import train\_test\_split**

**from sklearn.neighbors import KNeighborsClassifier**

**# Load and split data**

**X, y = make\_moons(noise=0.3, random\_state=0)**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(**

**X, y.astype(str), test\_size=0.25, random\_state=0)**

**trace\_specs = [**

**[X\_train, y\_train, '0', 'Train', 'square'],**

**[X\_train, y\_train, '1', 'Train', 'circle'],**

**[X\_test, y\_test, '0', 'Test', 'square-dot'],**

**[X\_test, y\_test, '1', 'Test', 'circle-dot']**

**]**

**fig = go.Figure(data=[**

**go.Scatter(**

**x=X[y==label, 0], y=X[y==label, 1],**

**name=f'{split} Split, Label {label}',**

**mode='markers', marker\_symbol=marker**

**)**

**for X, y, label, split, marker in trace\_specs**

**])**

**fig.update\_traces(**

**marker\_size=12, marker\_line\_width=1.5,**

**marker\_color="lightyellow"**

**)**

**fig.show()**

**Outpout:;**

**Q11)** **Develop a program to solve the eight queens problem. (Uninformed Search)**

**def is\_safe(board, row, col):**

**# Check the left side of the current row**

**for i in range(col):**

**if board[row][i] == 1:**

**return False**

**# Check upper left diagonal**

**for i, j in zip(range(row, -1, -1), range(col, -1, -1)):**

**if board[i][j] == 1:**

**return False**

**# Check lower left diagonal**

**for i, j in zip(range(row, len(board), 1), range(col, -1, -1)):**

**if board[i][j] == 1:**

**return False**

**return True**

**def solve\_queens(board, col):**

**if col >= len(board):**

**return True**

**for i in range(len(board)):**

**if is\_safe(board, i, col):**

**board[i][col] = 1**

**if solve\_queens(board, col + 1):**

**return True**

**board[i][col] = 0**

**return False**

**def print\_board(board):**

**for row in board:**

**print(" ".join(["Q" if cell == 1 else "." for cell in row]))**

**def eight\_queens():**

**n = 8 # Size of the chessboard**

**board = [[0 for \_ in range(n)] for \_ in range(n)]**

**if not solve\_queens(board, 0):**

**print("No solution exists.")**

**else:**

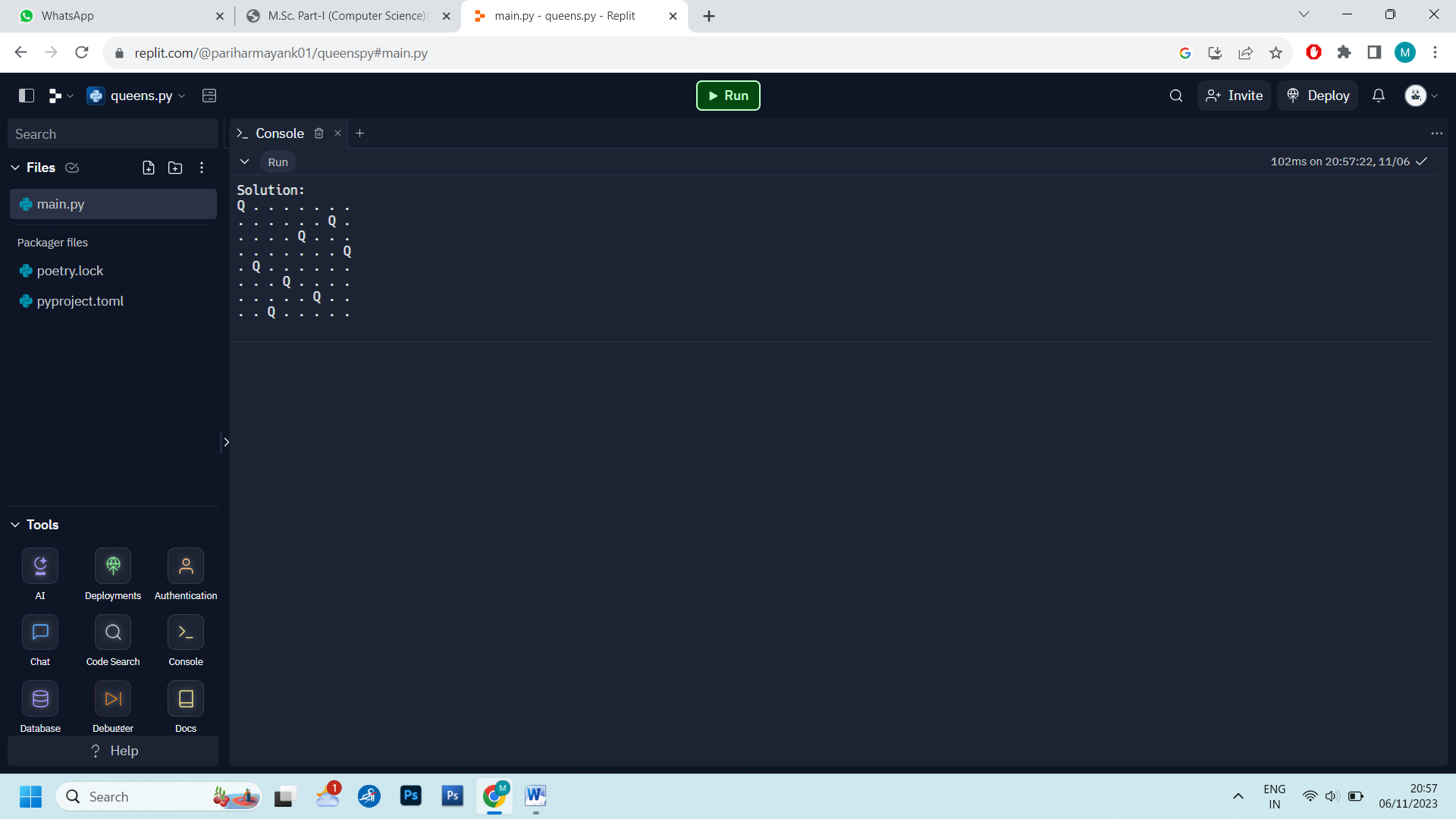
**print("Solution:")**

**print\_board(board)**

**if \_\_name\_\_== "\_\_main\_\_":**

**eight\_queens()**

Output:-



**Q14)Develop a program to solve the N queens puzzle using forward checking. Show in steps how the constraints are handled. (Constraint Satisfaction Problem)**

**Code:**

**def is\_safe(board, row, col):**

**# Check the left side of the current row**

**for i in range(col):**

**if board[row][i] == 1:**

**return False**

**# Check upper left diagonal**

**for i, j in zip(range(row, -1, -1), range(col, -1, -1)):**

**if board[i][j] == 1:**

**return False**

**# Check lower left diagonal**

**for i, j in zip(range(row, len(board), 1), range(col, -1, -1)):**

**if board[i][j] == 1:**

**return False**

**return True**

**def solve\_queens(board, col):**

**if col >= len(board):**

**return True**

**for i in range(len(board)):**

**if is\_safe(board, i, col):**

**board[i][col] = 1**

**if solve\_queens(board, col + 1):**

**return True**

**board[i][col] = 0**

**return False**

**def print\_board(board):**

**for row in board:**

**print(" ".join(["Q" if cell == 1 else "." for cell in row]))**

**def eight\_queens():**

**n = 8 # Size of the chessboard**

**board = [[0 for \_ in range(n)] for \_ in range(n)]**

**if not solve\_queens(board, 0):**

**print("No solution exists.")**

**else:**

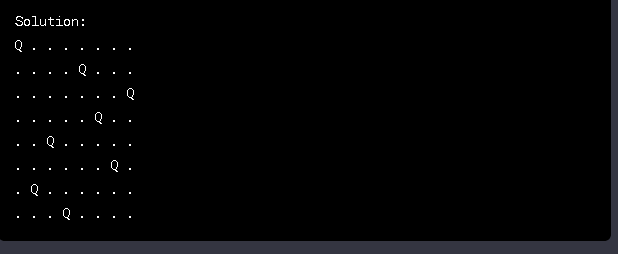
**print("Solution:")**

**print\_board(board)**

**if \_name\_ == "\_main\_":**

**eight\_queens()**

**output:**

****

**15)Write a computer program to play tic-tac-toe game. (Game Theory)**

**Code:**

**# Tic-Tac-Toe Game**

**# Initialize the board**

**board = [' ' for \_ in range(9)]**

**# Function to print the Tic-Tac-Toe board**

**def print\_board():**

**print(board[0] + '|' + board[1] + '|' + board[2])**

**print('-+-+-')**

**print(board[3] + '|' + board[4] + '|' + board[5])**

**print('-+-+-')**

**print(board[6] + '|' + board[7] + '|' + board[8])**

**# Function to check if the board is full**

**def is\_board\_full():**

**return ' ' not in board**

**# Function to check if a player has won**

**def check\_winner(player):**

**win\_combinations = [(0, 1, 2), (3, 4, 5), (6, 7, 8), (0, 3, 6), (1, 4, 7), (2, 5, 8), (0, 4, 8), (2, 4, 6)]**

**for combo in win\_combinations:**

**if board[combo[0]] == board[combo[1]] == board[combo[2]] == player:**

**return True**

**return False**

**# Function to get available moves**

**def get\_available\_moves():**

**return [i for i, space in enumerate(board) if space == ' ']**

**# Function to make a player's move**

**def make\_move(player, position):**

**board[position] = player**

**# Main game loop**

**current\_player = 'X'**

**while True:**

**print\_board()**

**print(f"Player {current\_player}'s turn. Enter your move (0-8): ")**

**move = int(input())**

**if move < 0 or move > 8 or board[move] != ' ':**

**print("Invalid move. Try again.")**

**continue**

**make\_move(current\_player, move)**

**if check\_winner(current\_player):**

**print\_board()**

**print(f"Player {current\_player} wins!")**

**break**

**elif is\_board\_full():**

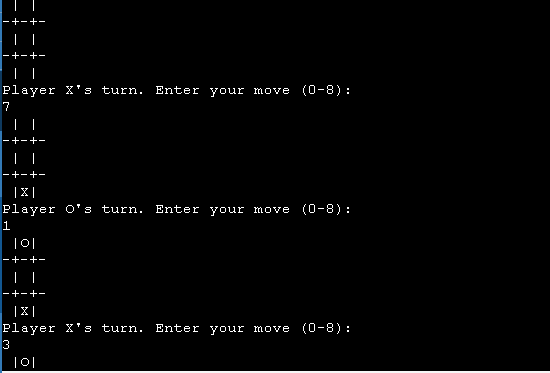
**print\_board()**

**print("It's a tie!")**

**break**

**current\_player = 'O' if current\_player == 'X' else 'X'**

**output::**

****