**SOURCE CODE:**

print("\t\tChatbox")

print("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")

while True:

a = input("Enter your question, please: ")

if a.lower() == "hi":

print("Hello!")

elif a.lower() == "how are you?":

print("I am fine. What about you?")

elif a.lower() == "what is your name?":

print("I am a simple chatbox created in Python.")

elif a.lower() == "i am fine too":

print("Thats good to hear.")

elif a.lower() == "is it raining?":

print("Yes it is.")

elif a.lower() == "bye" or a.lower() == "goodbye":

print("Goodbye! Have a great day!")

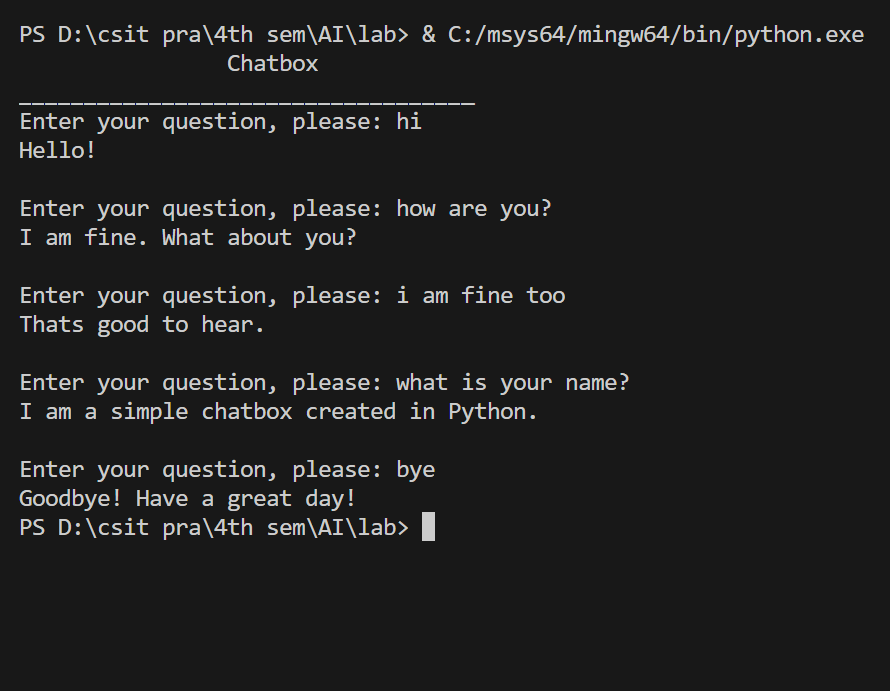
break

else:

print("Sorry, I don't understand that.")

print()

**OUTPUT:**

****

**SOURCE CODE:**

def print\_status(jug1, jug2):

print(f"Jug 1: {jug1} liters, Jug 2: {jug2} liters")

def water\_jug\_game():

jug1\_capacity = 4

jug2\_capacity = 3

jug1 = 0

jug2 = 0

print("Water Jug Problem Game")

print("Jug 1 capacity: 4 liters")

print("Jug 2 capacity: 3 liters")

print("Goal: Measure exactly 2 liters in one of the jugs.")

print("Commands: fill1, fill2, empty1, empty2, pour1to2, pour2to1, status, quit")

while True:

command = input("Enter command: ").strip().lower()

if command == "fill1":

jug1 = jug1\_capacity

elif command == "fill2":

jug2 = jug2\_capacity

elif command == "empty1":

jug1 = 0

elif command == "empty2":

jug2 = 0

elif command == "pour1to2":

transfer = min(jug1, jug2\_capacity - jug2)

jug1 -= transfer

jug2 += transfer

elif command == "pour2to1":

transfer = min(jug2, jug1\_capacity - jug1)

jug2 -= transfer

jug1 += transfer

elif command == "status":

print\_status(jug1, jug2)

elif command == "quit":

print("Exiting the game.")

break

else:

print("INVALID! Please try again.")

if jug1 == 2 or jug2 == 2:

print("Congratulations! You've solved water jug problem.")

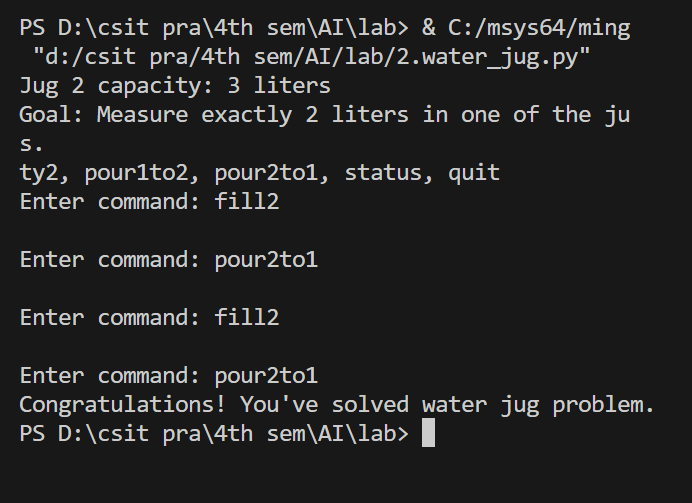
break

print()

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

water\_jug\_game()

**OUTPUT:**

****

**SOURCE CODE:**

def display(pegs):

print("\nCurrent State:")

for peg in pegs:

print(f"{peg}: {pegs[peg]}")

print()

def move(pegs, from\_peg, to\_peg):

if not pegs[from\_peg]:

print(f"Peg {from\_peg} is empty.")

return False

if pegs[to\_peg] and pegs[from\_peg][-1] > pegs[to\_peg][-1]:

print(f"Cannot move disk {pegs[from\_peg][-1]} to peg {to\_peg}.")

return False

pegs[to\_peg].append(pegs[from\_peg].pop())

return True

def is\_solved(pegs, ndisk, destinte):

return len(pegs[destinte]) == ndisk

def TOH(ndisk):

pegs = {'A': list(range(ndisk, 0, -1)), 'B': [], 'C': []}

print("Move the disks from peg A to peg C.")

while not is\_solved(pegs, ndisk, 'C'):

display(pegs)

command = input("Enter command: ").strip().lower()

if command == "quit":

break

parts = command.split()

if len(parts) == 3 and parts[0] == "move":

from\_peg, to\_peg = parts[1].upper(), parts[2].upper()

if from\_peg in pegs and to\_peg in pegs:

if move(pegs, from\_peg, to\_peg):

print(f"Moved disk from {from\_peg} to {to\_peg}.")

else:

print("Move failed.")

else:

print("Invalid pegs. Use A, B, or C.")

else:

print("Invalid command.")

if is\_solved(pegs, ndisk, 'C'):

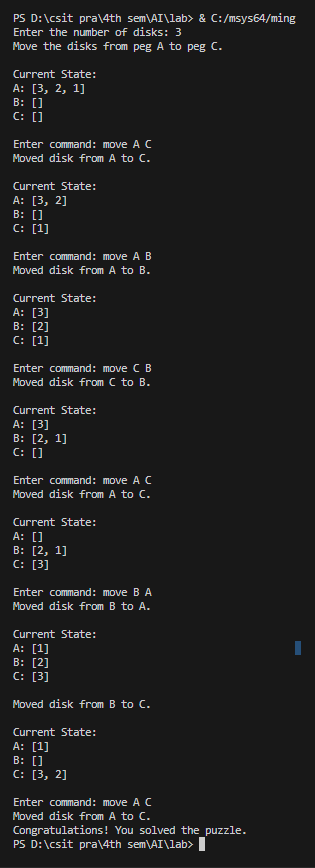
print("Congratulations! You solved the puzzle.")

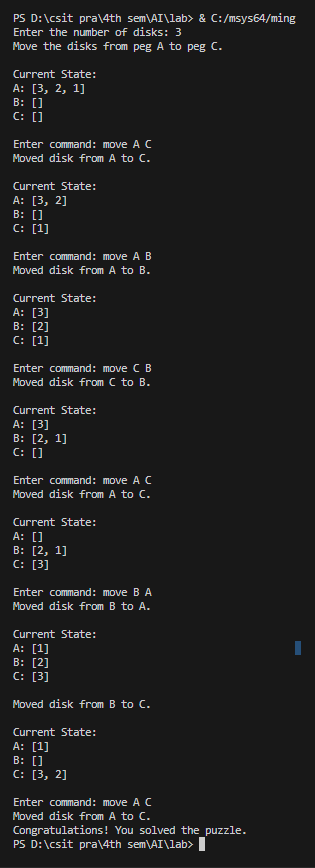
if \_\_name\_\_ == "\_\_main\_\_":

ndisk = int(input("Enter the number of disks: "))

TOH(ndisk)

**OUTPUT:**

****



**SOURCE CODE:**

def dfs\_graph(graph, start):

visited = set()

def dfs(node):

if node not in visited:

print(node)

visited.add(node)

for neighbor in graph[node]:

if neighbor not in visited:

dfs(neighbor)

dfs(start)

graph = {

'A': ['B', 'C'],

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

'C': ['A', 'F', 'G'],

'D': ['B', 'H'],

'E': ['B', 'I', 'J'],

'F': ['C', 'E'],

'G': ['C', 'I'],

'H': ['D', 'J'],

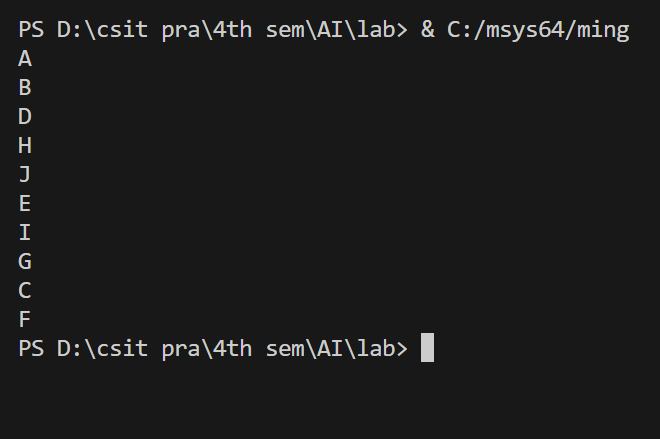
'I': ['E', 'G'],

'J': ['E', 'H']

}

dfs\_graph(graph, 'A')

**OUTPUT:**

****

**SOURCE CODE:**

from collections import deque

def bfs\_graph(graph, start):

visited = set()

queue = deque([start])

print("Route: ",end="")

while queue:

node = queue.popleft()

if node not in visited:

print(node,end=" ")

visited.add(node)

for neighbor in graph[node]:

if neighbor not in visited:

queue.append(neighbor)

graph = {

'A': ['B', 'C'],

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

'C': ['A', 'F', 'G'],

'D': ['B', 'H'],

'E': ['B', 'I', 'J'],

'F': ['C', 'E'],

'G': ['C', 'I'],

'H': ['D', 'J'],

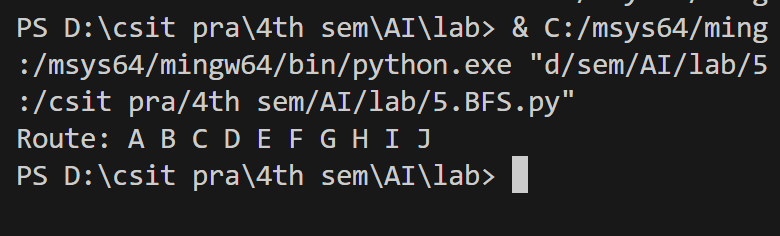
'I': ['E', 'G'],

'J': ['E', 'H']

}

bfs\_graph(graph, 'A')

**OUTPUT:**

****

**SOURCE CODE:**

import heapq

def heuristic(a, b):

return abs(a[0] - b[0]) + abs(a[1] - b[1])

def search(start, goal, graph):

open\_list = []

heapq.heappush(open\_list, (0 + heuristic(start, goal), 0, start))

came = {}

count = {start: 0}

while open\_list:

\_, current\_cost, current = heapq.heappop(open\_list)

if current == goal:

path = []

while current in came:

path.append(current)

current = came[current]

path.append(start)

return path[::-1]

for neighbor, cost in graph[current]:

new\_cost = count[current] + cost

if neighbor not in count or new\_cost < count[neighbor]:

count[neighbor] = new\_cost

priority = new\_cost + heuristic(goal, neighbor)

heapq.heappush(open\_list, (priority, new\_cost, neighbor))

came[neighbor] = current

return None

graph = {

(0, 0): [((0, 1), 1), ((1, 0), 1)],

(0, 1): [((0, 0), 1), ((1, 1), 1), ((0, 2), 1)],

(1, 0): [((0, 0), 1), ((1, 1), 1), ((2, 0), 1)],

(1, 1): [((0, 1), 1), ((1, 0), 1), ((1, 2), 1), ((2, 1), 1)],

(0, 2): [((0, 1), 1), ((1, 2), 1)],

(1, 2): [((0, 2), 1), ((1, 1), 1), ((2, 2), 1)],

(2, 0): [((1, 0), 1), ((2, 1), 1)],

(2, 1): [((1, 1), 1), ((2, 0), 1), ((2, 2), 1)],

(2, 2): [((1, 2), 1), ((2, 1), 1)]

}

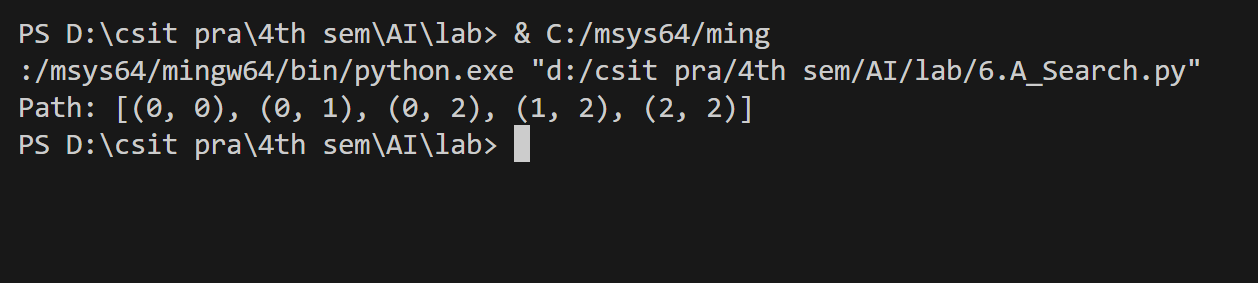
start = (0, 0)

goal = (2, 2)

path = search(start, goal, graph)

print("Path:", path)

**OUTPUT:**

****

**SOURCE CODE:**

import heapq

def heuristic(node, goal):

return 1

def best\_first\_search(graph, start, goal):

open\_list = []

heapq.heappush(open\_list, (heuristic(start, goal), start))

came\_from = {}

visited = set()

cost\_so\_far = {start: 0}

while open\_list:

\_, current = heapq.heappop(open\_list)

if current in visited:

continue

visited.add(current)

if current == goal:

path = []

while current in came\_from:

path.append(current)

current = came\_from[current]

path.append(start)

return path[::-1], cost\_so\_far[goal]

for neighbor, cost in graph[current]:

new\_cost = cost\_so\_far[current] + cost

if neighbor not in visited or new\_cost < cost\_so\_far[neighbor]:

cost\_so\_far[neighbor] = new\_cost

priority = heuristic(neighbor, goal)

heapq.heappush(open\_list, (priority, neighbor))

came\_from[neighbor] = current

return None, None

graph = {

'A': [('B', 1), ('C', 4)],

'B': [('A', 1), ('D', 2), ('E', 5)],

'C': [('A', 4), ('F', 3)],

'D': [('B', 2)],

'E': [('B', 5), ('H', 1)],

'F': [('C', 3)],

'G': [('C', 2), ('I', 6)],

'H': [('E', 1), ('J', 7)],

'I': [('G', 6)],

'J': [('H', 7)]

}

start = input("Enter the start node: ").upper()

goal = input("Enter the goal node: ").upper()

path, total\_cost = best\_first\_search(graph, start, goal)

if path:

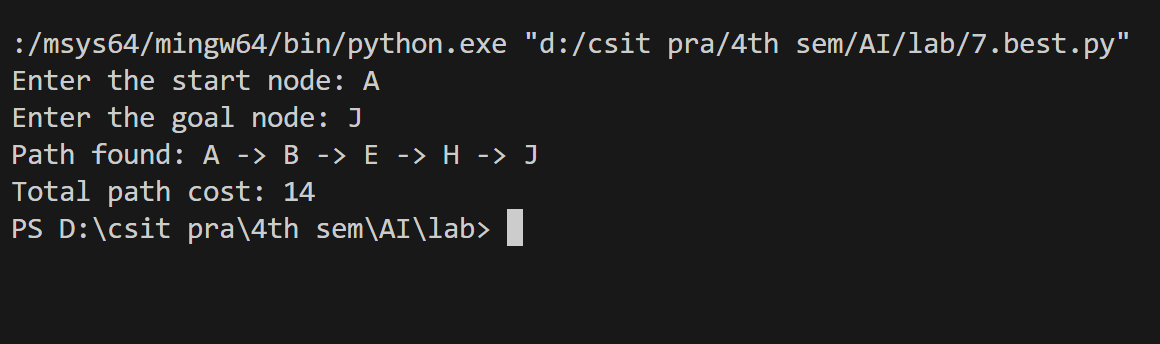
print("Path found:", " -> ".join(path))

print(f"Total path cost: {total\_cost}")

else:

print(f"No path found from {start} to {goal}")

**OUTPUT:**

****

**SOURCE CODE:**

import random

def get\_word():

    words = ["python", "hangman", "program", "computer", "science", "algorithm"]

    return random.choice(words).lower()

def display(word, guessed\_letters):

    return " ".join([letter if letter in guessed\_letters else "\_" for letter in word])

def hangman():

    print("Welcome to Hangman!")

    word = get\_word()

    guessed\_letters = set()

    attempts = 6

    while attempts > 0:

        print(f"\nWord: {display(word, guessed\_letters)}")

        print(f"Attempts remaining: {attempts}")

        print(f"Guessed letters: {' '.join(sorted(guessed\_letters)) if guessed\_letters else 'None'}")

        guess = input("Enter your guess (a single letter): ").lower()

        if len(guess) != 1 or not guess.isalpha():

            print("Invalid input. Please enter a single letter.")

            continue

        if guess in guessed\_letters:

            print("You've already guessed that letter. Try again.")

            continue

        guessed\_letters.add(guess)

        if guess in word:

            print("Correct!")

        else:

            print("Incorrect!")

            attempts -= 1

        if all(letter in guessed\_letters for letter in word):

            print(f"\nCongratulations! You guessed the word: {word}")

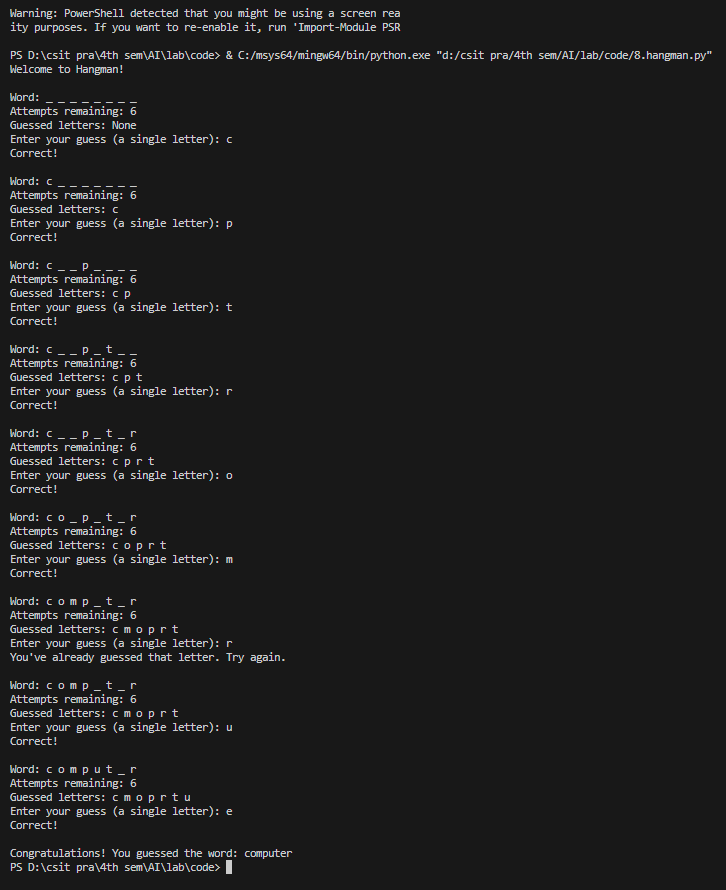
            break

    else:

        print(f"\nYou lost! The word was: {word}")

hangman()

**OUTPUT:**



**SOURCE CODE:**

from sklearn import metrics

from sklearn.naive\_bayes import GaussianNB

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

from sklearn.datasets import load\_iris

iris = load\_iris()

X = iris.data

y = iris.target

trX, teX, trY, teY = train\_test\_split(X, y, test\_size=0.6, random\_state=3)

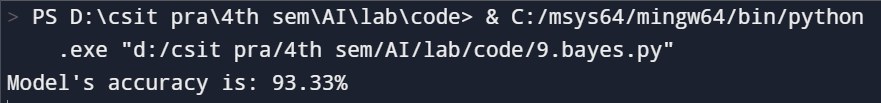
gnb = GaussianNB()

gnb.fit(trX, trY)

predictY = gnb.predict(teX)

print("Model's accuracy is: ", metrics.accuracy\_score(teY, predictY)\*100)

**OUTPUT:**



**SOURCE CODE:**

def print\_board(board):

    for row in board:

        print(" ".join(["Q" if x else "." for x in row]))

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

    for i in range(row):

        if board[i][col] == 1:

            return False

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

        if board[i][j] == 1:

            return False

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

        if board[i][j] == 1:

            return False

    return True

def solve\_nqueen(board, row, n):

    if row == n:

        print\_board(board)

        print("\nSolution found\n")

        return True

    for col in range(n):

        if is\_safe(board, row, col, n):

            board[row][col] = 1

            if solve\_nqueen(board, row + 1, n):

                return True

            board[row][col] = 0

return False

def n\_queen(n):

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

    if not solve\_nqueen(board, 0, n):

        print("No solution exists.")

try:

    n = int(input("Enter the size of the chessboard (N): "))

    if n < 4:

        print("There is no solution for N < 4.")

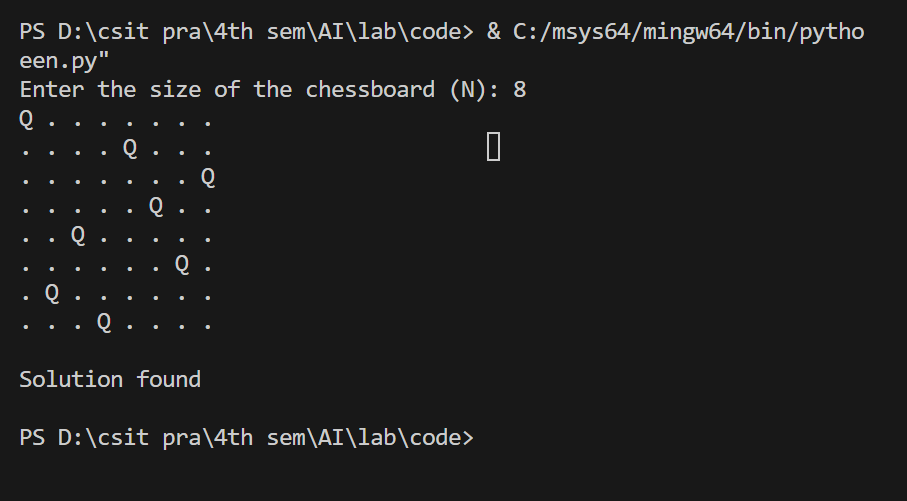
    else:

        n\_queen(n)

except ValueError:

    print("Please enter a valid integer.")

**OUTPUT:**



**SOURCE CODE:**

def get\_user\_input():

    print("Please answer the following questions with 'yes' or 'no'.")

    fever = input("Do you have a fever? (yes/no): ").lower() == 'yes'

    cough = input("Do you have a cough? (yes/no): ").lower() == 'yes'

    fatigue = input("Do you feel fatigued? (yes/no): ").lower() == 'yes'

    travel\_history = input("Have you recently traveled to a COVID-19 hotspot? (yes/no): ").lower() == 'yes'

    contact\_with\_positive = input("Have you been in contact with someone who has tested positive for COVID-19? (yes/no): ").lower() == 'yes'

    age = int(input("What is your age? "))

    chronic\_conditions = input("Do you have any chronic conditions (e.g., respiratory issues)? (yes/no): ").lower() == 'yes'

    return {

        "fever": fever,

        "cough": cough,

        "fatigue": fatigue,

        "travel\_history": travel\_history,

        "contact\_with\_positive": contact\_with\_positive,

        "age": age,

        "chronic\_conditions": chronic\_conditions

    }

def predict\_covid\_risk(user\_data):

    risk\_score = 0

    if user\_data['fever'] and user\_data['cough'] and user\_data['contact\_with\_positive']:

        risk\_score += 3

    if user\_data['travel\_history']:

        risk\_score += 2

    if user\_data['age'] > 65 and user\_data['chronic\_conditions']:

        risk\_score += 2

    if user\_data['fatigue'] and user\_data['fever']:

        risk\_score += 1

    if risk\_score >= 5:

        return "High Risk - Please consider getting tested for COVID-19."

    elif risk\_score >= 3:

        return "Moderate Risk - Monitor symptoms and consider testing."

    else:

        return "Low Risk - Continue observing health and take precautions."

def main():

    user\_data = get\_user\_input()

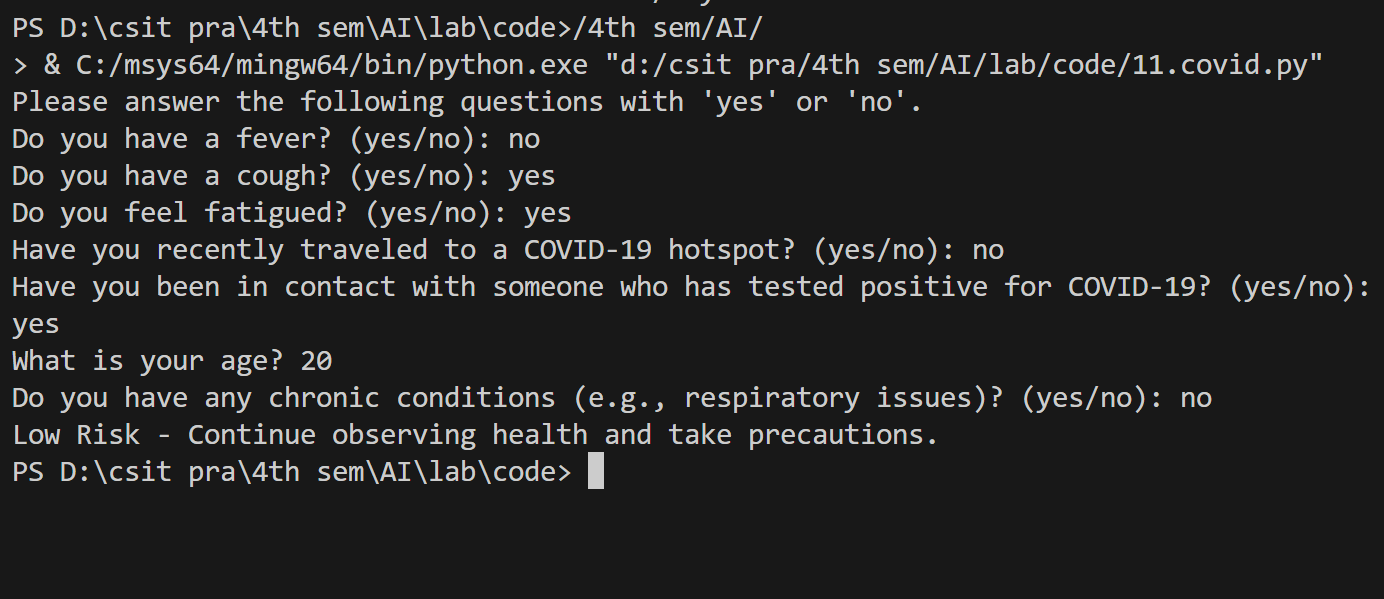
    result = predict\_covid\_risk(user\_data)

    print(result)

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

    main( )

**OUTPUT:**



**SOURCE CODE:**

from math import exp

def ANDGate(x1, x2):

stringLiteral = f"{x1} AND {x2} = "

weights = [-2, 1, 1]

Z = weights[0] + x1 \* weights[1] + x2 \* weights[2**] #linear combination**

sigmod\_val = 1 / (1 + exp(-Z)) **#sigmoid function**

if sigmod\_val >= 0.5:

print(stringLiteral + "1")

else:

print(stringLiteral + "0")

def ORGate(x1, x2):

stringLiteral = f"{x1} OR {x2} = "

weights = [-1, 2, 2]

Z = weights[0] + x1 \* weights[1] + x2 \* weights[2] **#linear combination**

sigmod\_val = 1 / (1 + exp(-Z)) **#sigmoid function**

if sigmod\_val >= 0.5:

print(stringLiteral + "1")

else:

print(stringLiteral + "0")

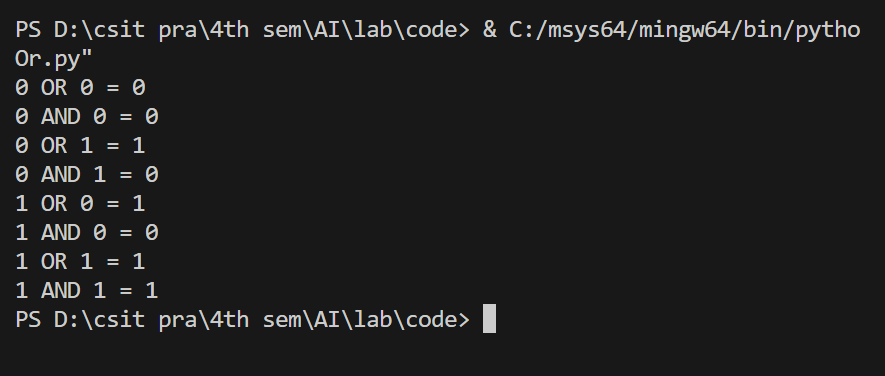
for x1 in range(2):

for x2 in range(2):

ORGate(x1, x2)

ANDGate(x1, x2)

**OUTPUT:**



**SOURCE CODE:**

def is\_valid(board, row, col, num):

    for i in range(9):   #check row

        if board[row][i] == num:

            return False

    for i in range(9):  **#check column**

        if board[i][col] == num:

            return False

    start\_row, start\_col = 3 \* (row // 3), 3 \* (col // 3)  **#check grid**

    for i in range(start\_row, start\_row + 3):

        for j in range(start\_col, start\_col + 3):

            if board[i][j] == num:

                return False

    return True

def solve\_sudoku(board):

**# Find the first empty cell (denoted by 0)**

    for row in range(9):

        for col in range(9):

            if board[row][col] == 0:

                for num in range(1, 10):

                    if is\_valid(board, row, col, num):

                        board[row][col] = num

                        if solve\_sudoku(board):

                            return True

                        board[row][col] = 0

                return False

    return True

def print\_board(board):

    for row in range(9):

        for col in range(9):

            print(board[row][col], end=" ")

        print()

board = [

    [5, 3, 0, 0, 7, 0, 0, 0, 0],

    [6, 0, 0, 1, 9, 5, 0, 0, 0],

    [0, 9, 8, 0, 0, 0, 0, 6, 0],

    [8, 0, 0, 0, 6, 0, 0, 0, 3],

    [4, 0, 0, 8, 0, 3, 0, 0, 1],

    [7, 0, 0, 0, 2, 0, 0, 0, 6],

    [0, 6, 0, 0, 0, 0, 2, 8, 0],

    [0, 0, 0, 4, 1, 9, 0, 0, 5],

    [0, 0, 0, 0, 8, 0, 0, 7, 9]

]

if solve\_sudoku(board):

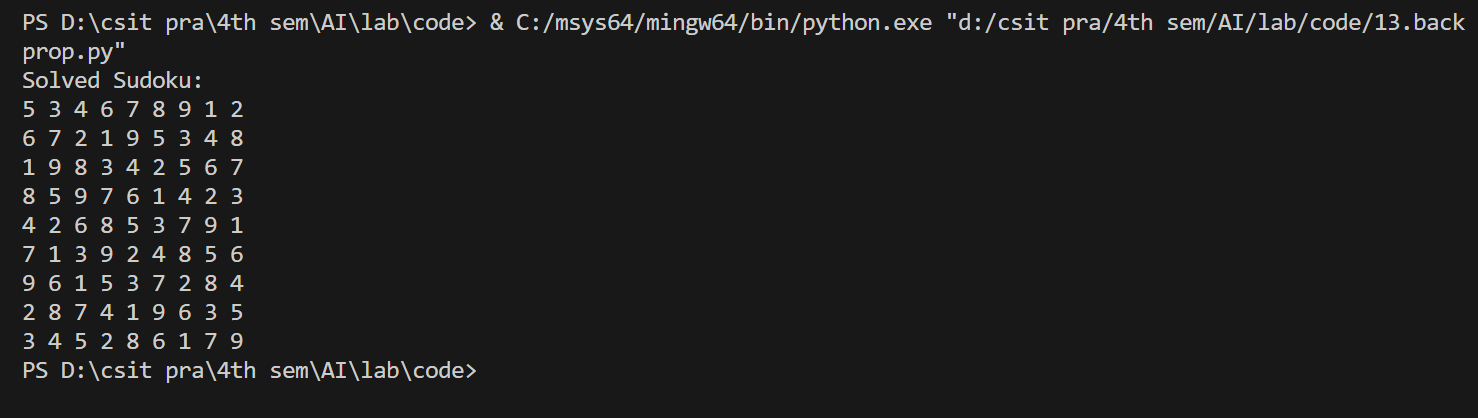
    print("Solved Sudoku:")

    print\_board(board)

else:

    print("No solution exists.")

**OUTPUT:**



**SOURCE CODE:**

def diaplay(board):

    for row in board:

        print(" ".join("Q" if cell else "." for cell in row))

    print("\n")

def safe(board, row, col, n):

    for i in range(col):

        if board[row][i]:

            return False

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

        if board[i][j]:

            return False

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

        if board[i][j]:

            return False

    return True

def solve(board, col, n, solutions):

    if col >= n:

        solutions.append([row[:] for row in board])

        return

    for i in range(n):

        if safe(board, i, col, n):

            board[i][col] = 1

            solve(board, col + 1, n, solutions)

            board[i][col] = 0

def solveQ(n):

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

    solutions = []

    solve(board, 0, n, solutions)

    print(f"Total solutions for {n}x{n} board: {len(solutions)}\n")

    for solution in solutions:

        diaplay(solution)

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

    try:

        N = int(input("Enter the value of N: "))

        if N <= 0:

            print("Please enter a positive integer.")

        else:

            solveQ(N)

    except ValueError:

        print("Invalid input. Please enter a positive integer.")

**OUTPUT:**

