1. Implement a simple reflex agent in Python to navigate a grid environment, avoiding obstacles and performing basic tasks.

import random, time

N = 5

grid = [['-' for \_ in range(N)] for \_ in range(N)]

# Place obstacles

for \_ in range(5):

    r, c = random.randint(0, N-1), random.randint(0, N-1)

    grid[r][c] = '#'

# Place tasks

for \_ in range(3):

    r, c = random.randint(0, N-1), random.randint(0, N-1)

    if grid[r][c] == '-':

        grid[r][c] = 'T'

# Place agent

agent = (0, 0)

collected = 0

def show():

    for row in grid:

        print(' '.join(row))

    print()

# Run simulation

for step in range(10):

    print(f"Step {step+1}:")

    r, c = agent

    # Collect task if present

    if grid[r][c] == 'T':

        collected += 1

        print(f"Task collected at location ({r},{c})! Total = {collected}")

    grid[r][c] = 'A'

    show()

    # Reflex move (random, avoids obstacles)

    moves = [(-1,0), (1,0), (0,-1), (0,1)]

    random.shuffle(moves)

    moved = False

    for dr, dc in moves:

        nr, nc = r+dr, c+dc

        if 0 <= nr < N and 0 <= nc < N and grid[nr][nc] != '#':

            grid[r][c] = '-' if grid[r][c]=='A' else grid[r][c]

            agent = (nr, nc)

            moved = True

            break

    if not moved:

        print("Agent cannot move! Surrounded by obstacles.")

    time.sleep(0.8)

print("Final Collected Tasks:", collected)

or

import time

N = 5

# Hardcoded grid: '-' empty, '#' obstacle, 'T' task

grid = [

    ['-', '-', 'T', '-', '-'],

    ['-', '#', '-', '-', '-'],

    ['-', '-', '-', 'T', '-'],

    ['T', '-', '#', '-', '-'],

    ['-', '-', '-', '-', '-']

]

# Place agent at top-left corner

agent = (0, 0)

collected = 0

def show():

    for row in grid:

        print(' '.join(row))

    print()

# Run simulation

for step in range(10):

    print(f"Step {step+1}:")

    r, c = agent

    # Collect task if present

    if grid[r][c] == 'T':

        collected += 1

        print(f"Task collected at location ({r},{c})! Total = {collected}")

    grid[r][c] = 'A'

    show()

    # Reflex move (random, avoids obstacles)

    moves = [(-1,0), (1,0), (0,-1), (0,1)]

    import random

    random.shuffle(moves)

    moved = False

    for dr, dc in moves:

        nr, nc = r+dr, c+dc

        if 0 <= nr < N and 0 <= nc < N and grid[nr][nc] != '#':

            grid[r][c] = '-' if grid[r][c]=='A' else grid[r][c]

            agent = (nr, nc)

            moved = True

          break

    if not moved:

        print("Agent cannot move! Surrounded by obstacles.")

    time.sleep(0.8)

print("Final Collected Tasks:", collected)

2 . Develop a game-playing agent in Python for Tic-Tac-Toe using the minimax algorithm to determine optimal moves.

import math

# Print the board

def print\_board(board):

    for row in [board[i\*3:(i+1)\*3] for i in range(3)]:

        print("| " + " | ".join(row) + " |")

    print()

# Check if a player has won

def winner(board, player):

    win\_states = [

        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # rows

        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # cols

        [0, 4, 8], [2, 4, 6]              # diagonals

    ]

    for state in win\_states:

        if all(board[i] == player for i in state):

            return True

    return False

# Check if board is full

def is\_full(board):

    return all(s != " " for s in board)

# Evaluate the board

def evaluate(board):

    if winner(board, "O"):  # AI is "O"

        return 1

    elif winner(board, "X"):  # Human is "X"

        return -1

    return 0

# Minimax algorithm

def minimax(board, depth, is\_maximizing):

    score = evaluate(board)

    # Terminal states

    if score == 1:

        return score

    if score == -1:

        return score

    if is\_full(board):

        return 0

    if is\_maximizing:  # AI turn (O)

        best\_score = -math.inf

        for i in range(9):

            if board[i] == " ":

                board[i] = "O"

                best\_score = max(best\_score, minimax(board, depth+1, False))

                board[i] = " "

        return best\_score

    else:  # Human turn (X)

        best\_score = math.inf

        for i in range(9):

            if board[i] == " ":

                board[i] = "X"

                best\_score = min(best\_score, minimax(board, depth+1, True))

                board[i] = " "

        return best\_score

# Find the best move for AI

def best\_move(board):

    best\_score = -math.inf

    move = None

    for i in range(9):

        if board[i] == " ":

            board[i] = "O"

            score = minimax(board, 0, False)

            board[i] = " "

            if score > best\_score:

                best\_score = score

                move = i

    return move

# Main game loop

def play\_game():

    board = [" "] \* 9

    print("Tic-Tac-Toe: You are X, AI is O")

    print\_board(board)

    while True:

        # Human move

        move = int(input("Enter your move (0-8): "))

        if board[move] != " ":

            print("Invalid move, try again.")

            continue

        board[move] = "X"

        print\_board(board)

        if winner(board, "X"):

            print("You win! 🎉")

            break

        if is\_full(board):

            print("It's a draw!")

            break

        # AI move

        ai\_move = best\_move(board)

        board[ai\_move] = "O"

        print("AI plays at:", ai\_move)

        print\_board(board)

        if winner(board, "O"):

            print("AI wins! 🤖")

            break

        if is\_full(board):

            print("It's a draw!")

            break

# Run the game

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

    play\_game()

or

import math

def print\_board(board):

for r in [board[i:i+3] for i in range(0,9,3)]:

print("|".join(r))

def winner(board):

wins = [(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 a,b,c in wins:

if board[a]==board[b]==board[c] and board[a]!=" ":

return board[a]

return None

def minimax(board, is\_max):

win = winner(board)

if win=="O": return 1

if win=="X": return -1

if " " not in board: return 0

scores=[]

for i in range(9):

if board[i]==" ":

board[i]="O" if is\_max else "X"

scores.append(minimax(board, not is\_max))

board[i]=" "

return max(scores) if is\_max else min(scores)

def best\_move(board):

best, move = -math.inf, None

for i in range(9):

if board[i]==" ":

board[i]="O"

score=minimax(board,False)

board[i]=" "

if score>best: best,move=score,i

return move

board=[" "]\*9

while True:

print\_board(board)

if winner(board) or " " not in board: break

board[int(input("Your move (0-8): "))]="X"

if winner(board) or " " not in board: break

board[best\_move(board)]="O"

3 Create a Python script to demonstrate the use of basic probability notation, including events, outcomes, and probability values

import random

# Sample space: all possible outcomes when rolling a fair 6-sided die

sample\_space = {1, 2, 3, 4, 5, 6}

print("Sample Space (S):", sample\_space)

# Define some events

event\_A = {2, 4, 6}   # Event A: rolling an even number

event\_B = {1, 2, 3}   # Event B: rolling a number <= 3

event\_C = {5}         # Event C: rolling exactly a 5

print("\nDefined Events:")

print("Event A (Even numbers):", event\_A)

print("Event B (Numbers <= 3):", event\_B)

print("Event C (Exactly 5):", event\_C)

# Probability of an event

def probability(event, sample\_space):

    return len(event) / len(sample\_space)

# Conditional probability P(A|B)

def conditional\_probability(A, B, sample\_space):

    if len(B) == 0:

        return 0

    return probability(A.intersection(B), sample\_space) / probability(B, sample\_space)

print("\nProbability Values:")

print("P(A) =", probability(event\_A, sample\_space))

print("P(B) =", probability(event\_B, sample\_space))

print("P(C) =", probability(event\_C, sample\_space))

# Intersection and Union (ASCII safe)

print("\nSet Operations:")

print("A INTERSECTION B =", event\_A.intersection(event\_B),

      " => P(A INTERSECTION B) =", probability(event\_A.intersection(event\_B), sample\_space))

print("A UNION B =", event\_A.union(event\_B),

      " => P(A UNION B) =", probability(event\_A.union(event\_B), sample\_space))

# Conditional probabilities

print("\nConditional Probabilities:")

print("P(A | B) =", conditional\_probability(event\_A, event\_B, sample\_space))

print("P(B | A) =", conditional\_probability(event\_B, event\_A, sample\_space))

4 Write a program to implement k-nearest Neighbor algorithm to classify the iris data set. Print both correct and wrong predictions

import pandas as pd

from sklearn.datasets import load\_iris

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

from sklearn.neighbors import KNeighborsClassifier

# Load Iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

target\_names = iris.target\_names

# Split dataset into train and test

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Create KNN model (k=3)

knn = KNeighborsClassifier(n\_neighbors=3)

knn.fit(X\_train, y\_train)

# Predictions

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

# Print results

print("Correct Predictions:")

for i in range(len(y\_test)):

if y\_test[i] == y\_pred[i]:

print(f"Sample {i}: True={target\_names[y\_test[i]]}, Predicted={target\_names[y\_pred[i]]}")

print("\nWrong Predictions:")

for i in range(len(y\_test)):

if y\_test[i] != y\_pred[i]:

print(f"Sample {i}: True={target\_names[y\_test[i]]}, Predicted={target\_names[y\_pred[i]]}")

# Accuracy

accuracy = (y\_test == y\_pred).sum() / len(y\_test)

print(f"\nAccuracy: {accuracy \* 100:.2f}%")