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}%")
```