```
#Task 1
import random
import math
def calculate_distance(point1, point2):
    x1, y1 = point1
    x2, y2 = point2
    distance = math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
    return distance
# Define the sensor function to perceive the location of the robot
def perceive_location():
    start = random.randint(1,locs)
    print("Robot starting at ",start)
    return start
# Define the rule-based agent function
def simple_reflex_agent(location,locs):
  if location == locs:
    location = 1
    print("Move camera towards location 1")
    pos = (random.randint(1, 10), random.randint(1, 10))
    print("Location ",location, " at coordinates ", pos)
    print("Distance between camera and location 1: ", calculate_distance(cam_loc,pos))
    return location
  else:
    location+=1
    print("Move camera towards location ",location)
    pos2 = (random.randint(1, 10), random.randint(1, 10))
    print("Location ",location, " at coordinates ", pos2)
    print("Distance between camera and location ", location, " : ", calculate_distance(cam_loc,pos2))
    return location
locs = int(input("Enter the number of locations"))
cam_loc = (4,0)
# Perceive the current location
current_location = perceive_location()
# Main loop of the agent
for x in range(locs):
    # Determine the action based on the current location
    current_location = simple_reflex_agent(current_location,locs)
```

```
Enter the number of locations7
Robot starting at  4
Move camera towards location  5
Location  5  at coordinates  (6, 10)
Distance between camera and location  5  :  10.198039027185569
Move camera towards location  6
Location  6  at coordinates  (9, 7)
Distance between camera and location  6  :  8.602325267042627
Move camera towards location  7
Location  7  at coordinates  (10, 7)
Distance between camera and location  7  :  9.219544457292887
Move camera towards location 1
Location  1  at coordinates  (10, 1)
Distance between camera and location 1:  6.082762530298219
Move camera towards location  2
Location  2  at coordinates  (7, 7)
Distance between camera and location  2  :  7.615773105863909
Move camera towards location  3
Location  3  at coordinates  (4, 1)
Distance between camera and location  3  :  1.0
Move camera towards location  4
Location  4  at coordinates  (5, 10)
Distance between camera and location  4  :  10.04987562112089
```

```
import random
import math

class Car:
    def __init__(self):
        self.front_camera_range = 8
        self.side_camera_range = 2
        self.rear_camera_range = 0.5
        self.current_lane = "middle"

    def detect_object(self, camera):
        # Simulate object detection within the camera range
        t = random.randint(0,math.ceil(camera+3))
        if t <camera:
          print("object detected at ",t, "meters with camera range",camera)
        return t < camera

    def apply_brakes(self):
        print("Brakes applied!")

    def move_to_left_lane(self):
        print("Moving to the left lane")

    def move_to_right_lane(self):
        print("Moving to the right lane")

    def parking(self):
        print("Parking...")
        self.apply_brakes()

    def operate_cameras(self):
        while True:
            # Simulate camera detections
            print("new simulation")
            if self.detect_object(self.front_camera_range):
                self.apply_brakes()
            elif self.detect_object(self.side_camera_range):

                if self.current_lane == "middle":
                    if self.detect_object(self.side_camera_range):

                        if random.choice([True, False]):
                            print("moving to left lane")
                            self.move_to_left_lane()
                        else:
                            print("moving to right lane")
                            self.move_to_right_lane()
            elif self.detect_object(self.rear_camera_range):
                self.parking()
                break
            else:
                print("No obstacles detected.")

# Create a car instance
car = Car()
# Operate the cameras
car.operate_cameras()
```

```
new simulation
object detected at  3 meters with camera range 8
Brakes applied!
new simulation
object detected at  1 meters with camera range 2
new simulation
object detected at  7 meters with camera range 8
Brakes applied!
new simulation
object detected at  7 meters with camera range 8
Brakes applied!
new simulation
object detected at  0 meters with camera range 8
Brakes applied!
new simulation
No obstacles detected.
new simulation
object detected at  7 meters with camera range 8
Brakes applied!
new simulation
```

```
object detected at  0 meters with camera range 2
new simulation
object detected at  2 meters with camera range 8
Brakes applied!
new simulation
object detected at  5 meters with camera range 8
Brakes applied!
new simulation
No obstacles detected.
new simulation
object detected at  0 meters with camera range 0.5
Parking...
Brakes applied!
```

```
#Task 3
# List of temperature data from the sensors in Celsius
temperature_data_celsius = [20, 22, 21, 23, 25, 24, 22, 23, 20]

# Convert Celsius to Fahrenheit for each temperature reading
temperature_data_fahrenheit = [(temp * 9/5) + 32 for temp in temperature_data_celsius]

# Calculate the average temperature in Fahrenheit
average_temperature_fahrenheit = sum(temperature_data_fahrenheit) / len(temperature_data_fahrenheit)

print("Average temperature in Fahrenheit:", average_temperature_fahrenheit)
```

```
Average temperature in Fahrenheit: 71.99999999999999
```

```
#Task4
import random
class VacuumCleaner:
    def __init__(self, n, m, room):
        self.n = n
        self.m = m
        self.room = room
        self.visited = set()
        self.directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
        self.current_position = (0, 0)  # Start from position (0, 0) by default

    def is_valid(self, x, y):
        return 0 <= x < self.n and 0 <= y < self.m

    def print_room(self):
        for row in self.room:
            print(" ".join(row))
        print()

    def clean(self, x, y):
        if self.room[x][y] == 'D':
            self.room[x][y] = 'C'

    def all_clean_around(self, x, y):
        for dx, dy in self.directions:
            nx, ny = x + dx, y + dy
            if self.is_valid(nx, ny) and self.room[nx][ny] == 'D':
                return False
        return True

    def move(self):
        self.visited.add(self.current_position)
        x, y = self.current_position
        self.clean(x, y)

        if self.all_clean_around(x, y):
            print("Surroundings are clean. Stopping.")
            return

        # Randomly choose a direction
        random.shuffle(self.directions)
        for dx, dy in self.directions:
            nx, ny = x + dx, y + dy
            if self.is_valid(nx, ny) and (nx, ny) not in self.visited:

                self.current_position = (nx, ny)
                print(self.current_position)
                self.move()
                break

# Define the dimensions of the room

n = int(input("Enter number of rows: "))
m = int(input("Enter number of columns: "))

# Define the initial status of the room (D for dirty, C for clean, B for blocked)
room = [['D' for _ in range(m)] for _ in range(n)]

# Assign random statuses to the room
for i in range(n):
    for j in range(m):
        status = random.choice(['D', 'C', 'B'])
        room[i][j] = status
# Create the vacuum cleaner object
vacuum_cleaner = VacuumCleaner(n, m, room)

print("Initial Room Status:")
vacuum_cleaner.print_room()

print("Vacuum Cleaner Path:")
vacuum_cleaner.move()
vacuum_cleaner.print_room()
```

```
Enter number of rows: 6
Enter number of columns: 7
```

```
Initial Room Status:
C D D C D C B
D D D D B B B
D C B C B C D
D B B B B C B
B D B C B C C
D D C C C D B

Vacuum Cleaner Path:
(1, 0)
(1, 1)
(1, 2)
(0, 2)
(0, 1)
Surroundings are clean. Stopping.
C C C C D C B
C C C D B B B
D C B C B C D
D B B B B C B
B D B C B C C
D D C C C D B
```

```
#Task5
import random
class TicTacToe:
    def __init__(self):
        self.board = [[' ' for _ in range(3)] for _ in range(3)]
        self.player_symbol = 'X'
        self.computer_symbol = 'O'

    def print_board(self):
        for row in self.board:
            print('|'.join(row))
            print('-' * 5)

    def check_winner(self, symbol):
        # Check rows and columns
        for i in range(3):
            if all(self.board[i][j] == symbol for j in range(3)) or \
                    all(self.board[j][i] == symbol for j in range(3)):
                return True

        # Check diagonals
        if all(self.board[i][i] == symbol for i in range(3)) or \
                all(self.board[i][2 - i] == symbol for i in range(3)):
            return True

        return False

    def is_full(self):
        return all(self.board[i][j] != ' ' for i in range(3) for j in range(3))

    def player_move(self, row, col):
        if self.board[row][col] == ' ':
            self.board[row][col] = self.player_symbol
            return True
        return False

    def computer_move(self):
        # Check if the computer can win
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == ' ':
                    self.board[i][j] = self.computer_symbol
                    if self.check_winner(self.computer_symbol):
                        return
                    self.board[i][j] = ' '

        # Check if the player can win and block them
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == ' ':
                    self.board[i][j] = self.player_symbol
                    if self.check_winner(self.player_symbol):
                        self.board[i][j] = self.computer_symbol
                        return
                    self.board[i][j] = ' '

        # Otherwise, make a random move
        while True:
            row = random.randint(0, 2)
            col = random.randint(0, 2)
            if self.board[row][col] == ' ':
                self.board[row][col] = self.computer_symbol
                return

    def play_game(self):
        print("Welcome to Tic Tac Toe!")
        self.print_board()

        while True:
            # Player's move
            row = int(input("Enter row (0, 1, or 2): "))
            col = int(input("Enter column (0, 1, or 2): "))
            if self.player_move(row, col):
                self.print_board()
                if self.check_winner(self.player_symbol):
                    print("Congratulations! You win!")
                    break
```

```
                elif self.is_full():
                    print("It's a draw!")
                    break

            # Computer's move
            print("Computer's move:")
            self.computer_move()
            self.print_board()
            if self.check_winner(self.computer_symbol):
                print("Computer wins!")
                break
            elif self.is_full():
                print("It's a draw!")
                break

# Create a Tic Tac Toe game instance and play the game
game = TicTacToe()
```

```
class PathfindingAgent:
    def __init__(self, grid, start, goal):
        self.grid = grid
        self.start = start
        self.goal = goal

    def manhattan_distance(self, point1, point2):
        return abs(point1[0] - point2[0]) + abs(point1[1] - point2[1])

    def is_valid_move(self, point):
        x, y = point
        return 0 <= x < len(self.grid) and 0 <= y < len(self.grid[0]) and self.grid[x][y] != 'X'

    def find_neighbors(self, point):
        x, y = point
        neighbors = [(x+1, y), (x-1, y), (x, y+1), (x, y-1)]
        valid_neighbors = [neighbor for neighbor in neighbors if self.is_valid_move(neighbor)]
        return valid_neighbors

    def utility(self, point):
        # Utility function based on Manhattan distance to the goal
        return -self.manhattan_distance(point, self.goal)

    def find_best_move(self, current_point):
        neighbors = self.find_neighbors(current_point)
        best_move = None
        max_utility = float('-inf')

        for neighbor in neighbors:
            neighbor_utility = self.utility(neighbor)
            if neighbor_utility > max_utility:
                best_move = neighbor
                max_utility = neighbor_utility
```