```
#Robot Localization done by pmurariu and baughboy
# [W,N,E,S]
# Correct Obstacle 0.90
# Correct Open Space 0.95
# Incorrect Open Space 0.10
# Incorrect Obstacle 0.05
import numpy as nump
maze = [
    [0,0,0,0,0,0,0],
    [0,1,0,1,0,1,0],
    [0,0,0,0,0,1,0],
    [0,0,0,1,0,0,0],
    [0,1,0,0,0,0,0],
    [0,1,0,1,0,1,0],
    [0,0,0,0,0,0,0],
    ]

#defining sensing probabiities
correct_obstacle = 0.90
correct_open_space = 0.95
incorrect_open_space = 0.10
incorrect_obstacle = 0.05

#defining the possible drift \
possible_drift = {
    'straight': 0.75,
    'left': 0.15,
    'right' : 0.10
    }

#making copy of the maze with location probabilties
rows, cols = len(maze), len(maze[0])
initial_probability = 0.025
prob_maze = nump.full((rows, cols), initial_probability)

def prediction(distance, action):
    new_distance = nump.zeros((7, 7), nump.float64) #array of zeroes
    for spaces in range(rows): #iterating
        for (state, prob) in transitional_prob(spaces, action): #iterate though
spaces we can travel to
            #add on term for total probability
            new_distance[state[0], state[1]] += prob * distance[spaces[0],
spaces[1]]
    return new_distance #update distribution

def transitional_prob(state, action):
    #go in intended direction
    drift_straight = transition(state, action)

    #left drift
    drift_left = transition(state, (action - 1) % 4)

    #drift right
    drift_right = transition(state, (action + 1) % 4)

    # return the 3 directions
    return ((drift_straight, possible_drift),
            (drift_left, possible_drift),
            (drift_right, possible_drift))
```

```
def transition(curr_state, action):
    global maze
    if (action == 0): #west
        new_location = (curr_state[0], curr_state[1] - 1)
    elif (action == 1): #north
        new_location = (curr_state[0] - 1, curr_state[1])
    elif (action == 2): #east
        new_location = (curr_state[0], curr_state[1] + 1)
    elif (action == 3): #south
        new_location = (curr_state[0] + 1, curr_state[1])

    if (new_location in maze or  # moving into obstacles
        new_location[0] < 0 or new_location[0] >= 7 or  #above or below maze
        new_location[1] < 0 or new_location[1] >= 7):  #left or right of maze
        return (curr_state[0], curr_state[1])  #stay where you are
    else:
        return new_location

#handles all of the moving and updating the prob maze after
def moving (move_direction):
    global prob_maze
    temp_prob_maze = nump.zeros_like(prob_maze)
    for row in range(rows):
        for col in range(cols):
            if maze[row][col] == 1:
                temp_prob_maze [row][col] = prob_maze[row][col]
            else:
                for row_change, col_change in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
                    #finding the new positions
                    new_pos_row, new_pos_col = row + row_change, col + col_change
                    #checking to see if the robot will bounce or not (not moving is
the same as bouncing for all intents and purposes)
                    if 0 <= new_pos_row < rows and 0 <= new_pos_col < cols:
                        #updating probabilties and checking for drift
                        if move_direction == 'E':
                            #checking for straight move
                            temp_prob_maze[new_pos_row][new_pos_col] +=
prob_maze[row][col] * possible_drift['straight']
                            #checking for the left drifft
                            left_drift_row = row - row_change
                            left_drift_col = col - col_change
                            temp_prob_maze[left_drift_row][left_drift_col] +=
prob_maze[row][col] * possible_drift['left']
                            #checking for the right drift
                            right_drift_row = row + row_change
                            right_drift_col = col + row_change
                            temp_prob_maze[right_drift_row][right_drift_col] +=
prob_maze[row][col] * possible_drift['right']
                        if move_direction == 'N':
                            #checking for straight move
                            temp_prob_maze[new_pos_row][new_pos_col] +=
prob_maze[row][col] * possible_drift['straight']
                            #checking for the left drifft
                            left_drift_row = row - row_change
                            left_drift_col = col - col_change
                            temp_prob_maze[left_drift_row][left_drift_col] +=
prob_maze[row][col] * possible_drift['left']
                            #checking for the right drift
```

```
                            right_drift_row = row + row_change
                            right_drift_col = col + row_change
                            temp_prob_maze[right_drift_row][right_drift_col] +=
prob_maze[row][col] * possible_drift['right']
                        if move_direction == 'W':
                            #checking for straight move
                            temp_prob_maze[new_pos_row][new_pos_col] +=
prob_maze[row][col] * possible_drift['straight']
                            #checking for the left drifft
                            left_drift_row = row - row_change
                            left_drift_col = col - col_change
                            temp_prob_maze[left_drift_row][left_drift_col] +=
prob_maze[row][col] * possible_drift['left']
                            #checking for the right drift
                            right_drift_row = row + row_change
                            right_drift_col = col + row_change
                            temp_prob_maze[right_drift_row][right_drift_col] +=
prob_maze[row][col] * possible_drift['right']
                        if move_direction == 'S':
                            #checking for straight move
                            temp_prob_maze[new_pos_row][new_pos_col] +=
prob_maze[row][col] * possible_drift['straight']
                            #checking for the left drifft
                            left_drift_row = row - row_change
                            left_drift_col = col - col_change
                            temp_prob_maze[left_drift_row][left_drift_col] +=
prob_maze[row][col] * possible_drift['left']
                            #checking for the right drift
                            right_drift_row = row + row_change
                            right_drift_col = col + row_change
                            temp_prob_maze[right_drift_row][right_drift_col] +=
prob_maze[row][col] * possible_drift['right']
    #replaces the values in the main prob maze with the tepm one
    prob_maze[:] = temp_prob_maze

def sensing(row, col):
    global maze
    direction = ["West", "North", "East","South"]
    result = []

    for dir in direction:
        if dir == "West":
            r,c = row, col - 1
        elif dir == "North":
            r,c = row - 1, col
        elif dir == "East":
            r,c = row, col + 1
        else:
            r,c = row +1, col

        if ( c < 0 or c>=7 or r >= 7 or r < 0):
         result.append(1)
        else:
         result.append(maze[r][c])
    return result

def filtering(visual, next_action):
    global prob_maze, rows, cols
    global correct_obstacle, incorrect_obstacle, correct_open_space,
```

```
incorrect_open_space

    #make a new prob maze to store updated probabilities
    new_prob_maze = [[0 for _ in range(cols)] for _ in range(rows)]

    for row in range(rows):
        for col in range(cols):
            if maze[row][col] == 0:
             current_prob = prob_maze[row][col]
             result = sensing(row, col)
             updated_prob = current_prob

             for v, r in zip(visual, result):
                if v == r:  #the visual matches the maze layout
                    updated_prob *= correct_obstacle if v == 1 else
correct_open_space
                else:  #the visual doesnt match the maze layout
                    updated_prob *= incorrect_obstacle if v == 1 else
incorrect_open_space

             new_prob_maze[row][col] = updated_prob
            else:
                continue

    new_prob_maze /= nump.sum(new_prob_maze)
    return new_prob_maze, "predict"

def maze_print(maze, prob_maze):
    for row in range(7):  # Added the missing 'for' keyword
        for col in range(7):
            if maze[row][col] == 1:  # Check if the cell is a wall
                print(f"({row},{col}): {0.00:.2f}%", end="\t")
            else:
                print(f"({row},{col}): {prob_maze[row][col] * 100:.2f}%", end="\t")
        print()  # Print a newline at the end of each row


#start of the maze
print ("Initial Probabilities")
maze_print(maze, prob_maze)

#list of actions to be performed as given by project instructions
actions_list = [
    #sensing
    ([0, 1, 0, 0], None),
    #prediction
    ([0, 0, 0, 0], 'E'),
    #sensing
    ([0, 0, 0, 0], None),
    #prediction
    ([0, 0, 0, 0], 'N'),
    #sensing
    ([1, 0, 0, 1], None),
    #move north
    ([0, 0, 0, 0], 'N'),
    #sensing
    ([0, 1, 0, 0], None),
    #moving west
    ([0, 0, 0, 0], 'W'),
```

```
    #sensing
    ([0, 1, 0, 1], None)
    ]

next_action = "filter"
#doing the actions
for visual, direction in actions_list:
    if next_action == "filter":
     prob_maze, next_action = filtering(visual, next_action)
    elif next_action == "predict":
        #make predict function
        #moving(direction)
        print("maze after action")
    maze_print(maze, prob_maze)
    print()

print("Programmed by pmurariu and baughboy")
```