FIT9136 Algorithms and Programming Foundations in Python

2023 Semester 2

Assignment 1

Student name: Shruthi Shashidhara Shastry
Student ID: 33684669
Creation date: 19-08-2023
Last modified date: 25-08-2023

In [1]:
```
# Libraries to import (if any)

# imported random as the game contains playing with computer

import random
```

3.1 Game menu function

In [2]:
```
# Implement code for 3.1 here

#function definition for game menu

def game_menu():
    print("!!!!!!!!!!!! WELCOME !!!!!!!!!!!!!!!")
    print("GOMOKU GAME MENU :")
    print("1.Start a Game")
    print("2.Print the Board")
```

```
    print("3.Place a Stone")
    print("4.Reset the Game")
    print("5.Exit")
```

In [3]:
```
# Test code for 3.1 here [The code in this cell should be commented]

#game_menu()

#output:
##When this function is called it displays the specified game menu
```

3.2 Creating the Board

In [4]:
```
# Implement code for 3.2 here

#function definition for creation of game board
#I have choosen list(used list comprehension) data structure for keeping track of the state of the board

def create_board(size):

    #as we are creating nXn size board so i am making use of range(size)in list comprehension
    return [[" " for each in range(size)] for each in range(size)]
```

In [5]:
```
# Test code for 3.2 here [The code in this cell should be commented]

#create_board(13)

#output:
#this will create the board of the game with the specified size(13 in this case) using list comprehension ie 13X13
```

3.3 Is the target position occupied?

In [6]:
```
 # Implement code for 3.3 here

#function definition to check whether the position is occupied or not
#x is the row_index and y is the col_index
# as mentioned in specification i have taken x and y as valid numeric indices so didnt test any conditions for that

def is_occupied(board,x,y):
```

```
    #the function is_occupied will return TRUE if the position is occupied or FALSE if the position is not occupied
    return board[x][y] != " "
```

In [7]:
```
# Test code for 3.3 here [The code in this cell should be commented]

#size = 9
#board = create_board(size)
#if is_occupied(board, 0,0):
#    print("Position that you have choosen is occupied!!")
#else:
#    print("Position that you have choosen is not occupied!!")


#output:
#Position that you have choosen is not occupied!!
```

In [8]:
```
# Test code for 3.3 here [The code in this cell should be commented]

#board = [['●', '', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', ' ', '●', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', ' ', ' ', ' ', '●', ' ', ' ', ' ' ],
#         [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', '●', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', ' ', ' ', '●', ' ', ' ', ' ', ' ' ]]

#if is_occupied(board, 0,0):
#    print("Position that you have choosen is occupied!!")
#else:
#    print("Position that you have choosen is not occupied!!")

#output:
#Position that you have choosen is occupied
```

Valid Position Function

In [9]:
```
#function definition for valid position

#this function is defined to check whether the position specified is valid or not i.e, within the specified range of board
```

```
def valid_position(board, x, y):
    #x and y are row and column index and it should be within 0 and len(board)which
    #is size specified for it to be a valid position
    return ((x>=0  and x<len(board)) and (y>=0 and y < len(board)))
```

In [10]:
```
# Test code for valid position here

#size = 9
#board = create_board(size)
#if valid_position(board,10,0):
#    print("Position that you have choosen is valid!!")
#else:
#    print("Position that you have choosen is not valid!!")

#output
#the size specificed is 9 and the position entered to place the stone is (10,0)
#which is not valid as its out of list index range
#so it prints Position that you have choosen is not valid!!
```

In [11]:
```
# Test code for valid position here

#size = 9
#board = create_board(size)
#if valid_position(board,8,0):
#    print("Position that you have choosen is valid!!")
#else:
#    print("Position that you have choosen is not valid!!")

#output
#Position that you have choosen is valid
```

3.4 Placing a Stone at a Specific Intersection

In [12]:
```
 # Implement code for 3.4 here

#function definition for placing a stone at a specific intersection
#the function place_on_board will return/output TRUE or FALSE indicating whether placing the stone was successful or not
def place_on_board(board, stone,position):
    #since position is tuple(x,y)
    x, y = position
```

```
    # Check if the position is valid i.e within the size of board and Check if the position is occupied or not
    if valid_position(board, x, y) and not is_occupied(board, x, y):
        board[x][y] = stone
        return True
    return False
```

In [13]:
```
# Test code for 3.4 here [The code in this cell should be commented]

#size = 9
#board = create_board(size)
#position = input("Enter the position:").split(" ")
#x = int(position[0])
#y= ord(position[1].upper())-ord("A")
#flag = place_on_board(board,"●",(x,y))
#if flag:
#    print("stone placed successfully")
#else:
#    print("error!! Invalid input")

#output:
#place_on_board returns true as the position is valid and not occupied and prints stone placed successfully.
```

In [14]:
```
# Test code for 3.4 here [The code in this cell should be commented]

#size = 9
#board = create_board(size)
#position = input("Enter the position:").split(" ")
#x = int(position[0])
#y= ord(position[1].upper())-ord("A")
#flag = place_on_board(board,"●",(x,y))
#if flag:
#    print("stone placed successfully")
#else:
#    print("error!! Invalid input")

#output
# place_on_board returns false as the position is invalid and prints error!! Invalid input
```

In [15]:
```
# Test code for 3.4 here [The code in this cell should be commented]

#board = [['●', '', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
```

```
#         [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', ' ', '●', ' ', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', ' ', ' ', ' ', '●', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#        [' ', '●', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#        [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', ' ', ' ', '●', ' ', ' ', ' ', ' ', ' ' ]]

#position = input("Enter the position:").split(" ")
#x = int(position[0])
#y= ord(position[1].upper())-ord("A")
#flag = place_on_board(board,"●",(x,y))
#if flag:
#    print("stone placed successfully")
#else:
#    print("error!! Invalid input or position is already occupied pls check!!")

#output
#place_on_board returns false as the position is occupied and prints
#error!! Invalid input or position is already occupied pls check!!
```

3.5 Printing the Board

In [16]:

```
# # Implement code for 3.5 here

#function definition for visualizing the board with -- as row and | as column grid
#along with row (0,1,2,...) and col(A,B,C....)indices

def print_board(board):

    size = len(board)

    # for printing column indices (A, B, C, ...)
    print("", end = "")
    for i in range(size):
        print("{}".format(chr(ord('A') + i)),end = "  ")
    print()

    # for printing the board rows( -- , stone at the intersection of row and column  and row index)
    print("", end = "")
    #loop for number of rows,
    #size-1 as x starts from 0
```

```
    for x in range(size-1):
        #loop for no of times -- to be printed in each row
        for z in range(size-1):
            #board[x][z] position where the stone has to be placed
            print("{}--".format(board[x][z]),end = "")
            #adding the stone position to be placed for last column in each row.
            if z == size-2:
                print("{}".format(board[x][z+1]),end = "")
        #printing row index at the end of each row
        print("",x)

        #for adding column grid
        for y in range(size-1):
            print("| " ,end=" ")
        print("|")

    #for printing the last row as in the above it prints only n-1 row so to
    #print the nth row grid ,stone and index we make use of same code of row
    print("", end = "")
    for x in range(size-1,size):
        for z in range(size-1):
            print("{}--".format(board[x][z]),end = "")
            if z == size-2:
                print("{}".format(board[x][z+1]),end = "")
        print("",x)
```

In [17]:
```
# Test code for 3.5 here [The code in this cell should be commented]


#board = create_board(9)
#position = input("Enter the position:\n").split(" ")
#x = int(position[0])
#y= ord(position[1].upper())-ord("A")
#place_on_board(board,"●",(x,y))
#print_board(board)

#output:
#this function will create and display the game board according to the specified size
#and displays stone at the specified intersection
```

3.6 Check Available Moves

In [18]:
```
 # Implement code for 3.6 here

#function definition to check the available moves
def check_available_moves(board):
    #return all tuples of the form(x,y) eg('0','F') if that position is not occupied
    return [(str(x), chr(y + ord("A"))) for x in range(len(board)) for y in range(len(board[0]))
            if not is_occupied(board, x, y)]
```

In [19]:
```
# Test code for 3.6 here [The code in this cell should be commented]

#board = create_board(9)
#available_moves = check_available_moves(board)
#print("No of Available Moves:",len(available_moves))
#print("Available Moves:", available_moves)

#output:
#displays list all 81 tuples which is the avaibable moves initially
#[('0', 'A'), ('0', 'B'), ('0', 'C'), ('0', 'D'), ('0', 'E'), ('0', 'F'), ('0', 'G'), ('0', 'H'), ('0', 'I'),
#('1', 'A'), ('1', 'B'), ('1', 'C'), ('1', 'D'), ('1', 'E'), ('1', 'F'), ('1', 'G'), ('1', 'H'), ('1', 'I'),
#('2', 'A'), ('2', 'B'), ('2', 'C'), ('2', 'D'), ('2', 'E'), ('2', 'F'), ('2', 'G'), ('2', 'H'), ('2', 'I'),
#('3', 'A'), ('3', 'B'), ('3', 'C'), ('3', 'D'), ('3', 'E'), ('3', 'F'), ('3', 'G'), ('3', 'H'), ('3', 'I'),
#('4', 'A'), ('4', 'B'), ('4', 'C'), ('4', 'D'), ('4', 'E'), ('4', 'F'), ('4', 'G'), ('4', 'H'), ('4', 'I'),
#('5', 'A'), ('5', 'B'), ('5', 'C'), ('5', 'D'), ('5', 'E'), ('5', 'F'), ('5', 'G'), ('5', 'H'), ('5', 'I'),
#('6', 'A'), ('6', 'B'), ('6', 'C'), ('6', 'D'), ('6', 'E'), ('6', 'F'), ('6', 'G'), ('6', 'H'), ('6', 'I'),
#('7', 'A'), ('7', 'B'), ('7', 'C'), ('7', 'D'), ('7', 'E'), ('7', 'F'), ('7', 'G'), ('7', 'H'), ('7', 'I'),
#('8', 'A'), ('8', 'B'), ('8', 'C'), ('8', 'D'), ('8', 'E'), ('8', 'F'), ('8', 'G'), ('8', 'H'), ('8', 'I')]
```

In [20]:
```
# Test code for 3.6 here [The code in this cell should be commented]

#board = [[' ', '●', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#          [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#          [' ', ' ', ' ', '●', ' ', ' ', ' ', ' ', ' ' ],
#          [' ', ' ', ' ', ' ', ' ', '●', ' ', ' ', ' ' ],
#          [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#          [' ', '●', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#          [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#          [' ', ' ', ' ', ' ', '●', ' ', ' ', ' ', ' ' ]]

#available_moves = check_available_moves(board)
#print("Available Moves:", len(available_moves))
```

```
#output
#76
```

3.7 Check for the Winner

In [1]:
```
 # Implement code for 3.7 here
#function definition for checking for winner
def check_for_winner(board):
    size = len(board)
    # Black and White stones
    players = ["●", "○"]

    for player in players:
        #for row index
        for x in range(size):
            #for col index
            for y in range(size):
                stone = board[x][y]
                if stone == player:
                    # Check horizontally
                    #only col value changes
                    if y + 4 < size and all(board[x][y+i] == player for i in range(5)):
                        return player
                    # Check vertically
                    #only row value changes
                    if x + 4 < size and all(board[x+i][y] == player for i in range(5)):
                        return player
                    # Check diagonal
                    #both bottom-left to top-right and top-left to bottom-right
                    if x + 4 < size and y + 4 < size and all(board[x+i][y+i] == player for i in range(5)):
                        return player
                    # Check anti-diagonal
                    #both bottom-right to top-left and top-right to bottom-left
                    if x + 4 < size and y >= 4 and all(board[x+i][y-i] == player for i in range(5)):
                        return player
    #check for draw
    if len(check_available_moves(board)) == 0:
        return "Draw"
    else:
        return None
```

In [22]:
```
# Test code for 3.7 here [The code in this cell should be commented]

#board1 = [[' ', '●', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', '●', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#         ['○', '○', '○', '●', ' ', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', '○', ' ', '●', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', ' ', '○', ' ', '●', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', '', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '  ],
#         [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ' ],
#         [' ', ' ', ' ', ' ', '', ' ', ' ', ' ', ' ', ' ' ]]



#winner = check_for_winner(board)
#if winner :
#    if winner == "Draw":
#        print("\nIt's a draw!")
#    else:
#        print(f"{winner} wins!!" )

#output
# ● wins!!
```

In [23]:
```
# Test code for 3.7 here [The code in this cell should be commented]

#board1 = [['●', '●', '○', '●', '○', '●'],
#          ['●', '●', '●', '●', '○', '●'],
#          ['○', '●', '○', '●', '○', '●'],
#          ['○', '○', '●', '●', '●', '○'],
#          ['○', '●', '○', '○', '○', '○'],
#          ['○', '○', '○', '●','○', '○']]

#winner = check_for_winner(board1)
#if winner :
#    if winner == "Draw":
#        print("\nIt's a draw!")
#    else:
#        print(f"{winner} wins!!" )

#output
#It's a draw
```

3.8 Random Computer Player

In [2]:
```
 # Implement code for 3.8 here
#function definition for random computer player
def random_computer_player(board, player_move):
     #choose a random available position,if its computer's first move
    if player_move is None:
        return random.choice(check_available_moves(board))

    x, y = player_move

    #to find all the available valid positions within a 3 * 3 square
    valid_positions = []
    for i in range(-1, 2):
        for j in range(-1, 2):
            new_x, new_y = x + i, y + j
            # Check if the new position is valid and unoccupied
            if valid_position(board, new_x, new_y) and not is_occupied(board, new_x, new_y):
                valid_positions.append((new_x, new_y))

    #If there are no valid positions around the player's move, choose a random available position
    if not valid_positions:
        random_tuple = random.choice(check_available_moves(board))
        # Convert the string digit part of the tuple to its row index
        cx = int(random_tuple[0])
         # Convert the letter part of the tuple to its corresponding column index
        cy = ord(random_tuple[1].upper())- ord("A")
        return (cx,cy)
    else:
        return random.choice(valid_positions)
```

In [25]:
```
# Test code for 3.8 here [The code in this cell should be commented]

#player_move = (2,'B')
#x, y = player_move
#x = int(player_move[0])
#y = ord(player_move[1].upper())- ord("A")
#board = create_board(9)
#computer_move = random_computer_player(board,(x,y))
#print(f"Computer's Move: {computer_move[0]} {chr(computer_move[1] + ord('A'))}")
```

```
#output i got : but it can be any random value
#Computer's Move: 2 A
```

3.9 Play Game

In [26]:
```
# Implement code for 3.9 here

def play_game():
    #Initialize default values
    size = 9
    mode = "Player vs. Player"
    board = create_board(size)
    player_turn = "●"
    game_in_progress = False
    computer_has_played = False

    # Main game loop
    while True:

        #displaying game menu
        game_menu()


        choice = input("Enter your choice: ")

        # Option 1 - Start a New Game
        if choice == "1":
            # Check if a game is already in progress
            if game_in_progress:
                print("A game is already in progress.")
                reset_choice = input("1. Reset and start a new game\n2. Continue and complete the current game"
                                     +"\nEnter your choice: ")
                #reset and restart the game
                if reset_choice == '1':

                    game_in_progress = False

                    while True:
                        size = input("Enter board size (e.g. 9, 13, 15): ")
                        if size.isdigit():
                            size = int(size)
                            if size < 6:
```

```
                            print("\nERROR: Enter a valid board size (e.g., 9, 13, 15...)\n")
                        else:
                            #Exit the loop if a valid size is entered
                            break
                    else:
                        print("\nERROR: Please enter a valid numeric value for board size!!!\n")
                while True:
                    mode = input("Enter mode (Player vs. Player (PVP) / Player vs. Computer (PVC)): ").lower()
                    if mode in ['pvp', 'pvc', 'player vs. player', 'player vs. computer']:
                     # Exit the loop if a valid mode is entered
                        break
                    else:
                        print("\nINVALID MODE! Please enter a valid mode (PVP/PVC).\n")

                print("New game started.")
                board = create_board(size)
                player_turn = "●"
                game_in_progress = True
            else:
                game_in_progress = True
                print("Continuing the current game.")


        else:
            # Start a new game with user-defined settings
            #checking validation
            while True:
                size = input("Enter board size (e.g. 9, 13, 15): ")
                if size.isdigit():
                    size = int(size)
                    if size < 6:
                        print("\nERROR: Enter a valid board size (e.g., 9, 13, 15...)\n")
                    else:
                        #Exit the loop if a valid size is entered
                        break
                else:
                    print("\nERROR: Please enter a valid numeric value for board size!!!\n")
            #checking validation
            while True:
                mode = input("Enter mode (Player vs. Player (PVP) / Player vs. Computer (PVC)): ").lower()
                if mode in ['pvp', 'pvc', 'player vs. player', 'player vs. computer']:
                     # Exit the loop if a valid mode is entered
                    break
                else:
```

```
                    print("\nINVALID MODE! Please enter a valid mode (PVP/PVC).\n")

            game_in_progress = True
            print("\nGAME STARTED !! ALL THE BEST :)\n")
            board = create_board(size)
            player_turn = "●"

        # Option 2 - Print the Board
        elif choice == "2":
            print("\nTHE CURRENT STATE OF THE BOARD IS AS SHOWN BELOW:\n")
            print_board(board)
            print(" ")

        # Option 3 - Place a Stone
        elif choice == "3":
            print("\n")

             # Check if a game is in progress
            if not game_in_progress:
                print("No game in progress. Start a new game first.")
                continue
             # Player's turn to play
            print(f"{player_turn} turn to play")

            # Handle computer's move in PVC mode
            if mode in ["Player vs. Computer", "pvc"] and player_turn == "○":
                if not computer_has_played:
                    # Computer's first move
                    # None as the second argument
                    computer_move = random_computer_player(board, None)
                    computer_has_played = True
                else:
                    # Computer's subsequent move
                    computer_move = random_computer_player(board, (x, y))
                cx, cy = computer_move  # Extract row and column from the tuple
                cx = int(cx)
                cy = ord(cy.upper()) - ord("A")
                column_char = chr(cy + ord('A'))
                place_on_board(board, "○", (cx, cy))
                print(f"Computer's Move: {cx} {column_char}\n")
                print_board(board)
                player_turn = "●"

            # Player's move
```

```
        while True:
            position = input("\nEnter position to place stone (e.g. '2 F'): ").split()
            if len(position) >= 2:
                parts = position
                x, y = parts
                if x.isdigit():
                    x = int(x)
                    if not (0 <= x < len(board)):
                        print("Invalid row index. Please enter a valid numeric value.")
                        continue
                else:
                    print("Invalid input for row. Please enter a valid numeric value.")
                    continue
                #checking validation
                if y.isalpha():
                    y = ord(y.upper()) - ord("A")
                    if not (0 <= y < len(board)):
                        print("Invalid column index. Please enter a valid column character.")
                        continue
                else:
                    print("Invalid input for column. Please enter a valid column character.")
                    continue

                if is_occupied(board, x, y):
                    print("\nPosition already occupied. Try again.\n")
                    continue
                else:
                    if place_on_board(board, player_turn, (x, y)):
                        print("\n")
                        print_board(board)
                        winner = check_for_winner(board)
                        if winner:
                            print_board(board)
                            if winner == "Draw":
                                print("\nIt's a draw!")
                            else:
                                print(f"{winner} wins! CONGRATULATIONS :) ")
                            print("\nWhat do you want to do next?")
                            game_in_progress = False
                            break
                        else:
                            player_turn = "o" if player_turn == "●" else "●"
                            if mode in ["Player vs. Computer", "pvc"] and player_turn == "o":
                                computer_move = random_computer_player(board, (x, y))
```

```
                                place_on_board(board, player_turn, computer_move)
                                winner = check_for_winner(board)
                                if winner:
                                    print_board(board)
                                    if winner == "Draw":
                                        print("It's a draw!")
                                    else:
                                        print(f"{winner} wins!")
                                    game_in_progress = False
                                else:
                                    print(f"Computer's Move: {computer_move[0]} {chr(computer_move[1] + ord('A'))}\n")
                                    print_board(board)
                                    player_turn = "●"
                            break
                    else:
                        print("\nStone could not be placed due to Invalid or occupied position. Try again.\n")
            else:
                print("\nInvalid input. Please enter both row and column values for placing the stone.\n")
                continue

        # Option 4 - Reset the Game
        elif choice == "4":
            #reset and restart the game
            game_in_progress = False

            while True:
                size = input("Enter board size (e.g. 9, 13, 15): ")
                if size.isdigit():
                    size = int(size)
                    if size < 6:
                        print("\nERROR: Enter a valid board size (e.g., 9, 13, 15...)\n")
                    else:
                         # Exit the loop if a valid size is entered
                        break
                else:
                    print("\nERROR: Please enter a valid numeric value for board size!!!\n")
            while True:
                mode = input("Enter mode (Player vs. Player (PVP) / Player vs. Computer (PVC)): ").lower()
                if mode in ['pvp', 'pvc', 'player vs. player', 'player vs. computer']:
                         # Exit the loop if a valid mode is entered
                    break
                else:
                    print("\nINVALID MODE! Please enter a valid mode (PVP/PVC).\n")
```

```
            print("Game got reset.")
            board = create_board(size)
            player_turn = "●"
            game_in_progress = True
        # Option 5 - exit the Game
        elif choice == "5":
            print("\nTHANK YOU \nSEE YOU SOON\nEXITING THE GAME...\n")
            break

        else:
            print("Invalid choice. Try again.")
```

In [27]:
```
# Test code for 3.9 here [The code in this cell should be commented]
```

In [28]:
```
#play_game()
```

In [ ]:
```

```

Documentation of Optimizations

If you have implemented any optimizations in the above program, please include a list of these optimizations along with a brief explanation for each in this section.

1) The game logic has been enhanced to ensure an engaging and user-friendly experience. After each move, whether by the user or the computer, the game board is displayed to provide a visual representation of the ongoing game. 2) To streamline the code, list comprehensions have been employed in the functions create_board(size) and check_available_moves(board) for creating the game board and determining available moves, respectively. 3) The code for checking a winning condition when five stones of the same color are aligned has been optimized. Common loop now handles all four directions of possible alignments: horizontal, vertical, diagonal (both bottom-left to top-right and top-left to bottom-right), and anti-diagonal (both bottom-right to top-left and top-right to bottom-left).
4) A function to validate positions(valid_position()) has been implemented to ascertain whether the specified position falls within the dimensions of the game board. 5) Incorporated input validation within the play_game() function to prevent errors and interruptions caused by user inputs.

--- End of Assignment 1 ---