FIT9136 Algorithms and Programming Foundations in Python

2023 Semester 2

Assignment 1

```
**Student name:** Subbulakshmi Natarajan </br>
**Student ID:** 34069178 </br>
**Creation date:**  21 August 2023 </br>
**Last modified date:** 12:00pm </br>
```

In [1]:

```python
# Libraries to import (if any)
import random
```

3.1 Game menu function

In [3]:

```
# Implement code for 3.1 here

# we need to first print the game menu in order to make sure that user is able to
print("1. Start a game")
print("2. Print the board")
print("3. Place a stone")
print("4. Reset the game")
print("5. Exit")

# After the menu is printed the user can input the menu options he/she wants to se
game_user = input("Enter your menu that you want to choose in the game:")

# using the def function we can use combine loop for the creating the menu functio
def game_menu():

    if game_user == '1':
        print("Start a game")
    elif game_user == '2':
        print("Print the board")
    elif game_user == '3':
        print("Place a stone")
    elif game_user == '4':
        print("Reset the game")
    elif game_user == '5':
        print("Exit")
    else:
        print("Invalid option. Please select a number from 1 to 5.")

#you print the game_menu() as you used the def function in the previous code inste
game_menu()
```

```
1. Start a game
2. Print the board
3. Place a stone
4. Reset the game
5. Exit
Enter your menu that you want to choose in the game:1
Start a game
```

In [4]:

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

# we need to first print the game menu in order to make sure that user is able to
#print("1. Start a game")
#print("2. Print the board")
#print("3. Place a stone")
#print("4. Reset the game")
#print("5. Exit")

# After the menu is printed the user can input the menu options he/she wants to se

#game_user = input("Enter your menu that you want to choose in the game:")

# using the def function we can use combine loop for the creating the menu functic
#so that we can program the menu as per given in the project
#for example when the user enter the input in the game menu function then either t

#def game_menu():

    #if game_user == '1':
        #print("Start a game")
    #elif game_user == '2':
        #print("Print the board")
    #elif game_user == '3':
        #print("Place a stone")
    #elif game_user == '4':
        #print("Reset the game")
    #elif game_user == '5':
        #print("Exit")
    #else:
        #print("Invalid option. Please select a number from 1 to 5.")

#you print the game_menu() as you used the def function in the previous code inste

#game_menu()
```

3.2 Creating the Board

In [5]:

```
# Implement code for 3.2 here

#This line assigns the value 9 to the variable size, indicating the dimensions of
#In this case I have coded as 9 rows and 9 columns as per the question in the give

size = 9

#I created a function that is named as create_board that takes a single parameter
#Under the def functiomn I have used a nested list comprehension to generate a 2D
#The inner list comprehension [" " for _ in range(size)] creates a row of spaces (
#The outer list comprehension [for _ in range(size)] - this code creates row creat

def create_board(size):
    return [[" " for _ in range(size)] for _ in range(size)]

#usage of the code whether the function is working


board = create_board(size)
board
```

Out[5]:

```
[[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']]
```

In [6]:

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

#This line assigns the value 9 to the variable size, indicating the dimensions of
#In this case I have coded as 9 rows and 9 columns as per the question in the give

#size = 9

#I created a function that is named as create_board that takes a single parameter
#Under the def functiomn I have used a nested list comprehension to generate a 2D
#The inner list comprehension [" " for _ in range(size)] creates a row of spaces (
#The outer list comprehension [for _ in range(size)] - this code creates row creat

#def create_board(size):
    #return [[" " for _ in range(size)] for _ in range(size)]

#usage of the code whether the function is working


#board = create_board(size)
#board
```

3.3 Is the target position occupied?

In [ ]:

```
#### Implement code for 3.333 here

#The purpose of this function is to check if a specific cell on the game board is
#This line calculates the size of the game board.
#Since the game board is a square 2D list, its size is the number of rows or colum
#The len(board) function returns the number of rows in the board and this value is

def is_occupied(board, x, y):
    size = len(board)
    #0 <= x < size This condition checks whether x is within the range of valid ro

    if 0 <= x < size and 0 <= ord(y) - ord('A') < size:

        return board[x][ord(y) - ord('A')] != " " #This condition checks whether t
    return False


# Testing the function if it works
size = 9 # size of the board
board = create_board(size) # using the same function to create the board
row = 0 # indicating the rows
column = ord('A') - ord('A')  # Convert 'A' to column index
board[row][column] = "occupied"  # Occupying the cell at (0, 'A')

#Try to print the function with the parameter
print(is_occupied(board, 0, 'A'))  # Should print True
print(is_occupied(board, 3, 'A'))  # Should print False
```

In [8]:

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

#The purpose of this function is to check if a specific cell on the game board is
#This line calculates the size of the game board.
#Since the game board is a square 2D list, its size is the number of rows or colum
#The len(board) function returns the number of rows in the board and this value is

#def is_occupied(board, x, y):
    #size = len(board)
    #0 <= x < size This condition checks whether x is within the range of valid ro
    #0 <= ord(y) - ord('A') < size: This condition checks whether the ASCII value

    #if 0 <= x < size and 0 <= ord(y) - ord('A') < size:

        #return board[x][ord(y) - ord('A')] != " "
    #return False


# Testing the function if it works
#size = 9 # size of the board
#board = create_board(size) # using the same function to create the board
#row = 0 # indicating the rows
#column = ord('A') - ord('A')  # Convert 'A' to column index
#board[row][column] = "occupied"  # Occupying the cell at (0, 'A')

#Try to print the function with the parameter
#print(is_occupied(board, 0, 'A'))  # Should print True
#print(is_occupied(board, 3, 'A'))  # Should print False
```

3.4 Placing a Stone at a Specific Intersection

In [9]:

```
 # Implement code for 3.4 here
# we need to include the two other def function that we created previously(ie. cre
def create_board(size):
    return [[" " for _ in range(size)] for _ in range(size)]

def is_occupied(board, x, y):
    size = len(board)
    if 0 <= x < size and 0 <= y < size:
        return board[x][y] != " "
    return False


# First Define a function to place a stone on the board which has three parameter
def place_on_board(board, stone, position):
    x, y_char = position  # Extract row index (x) and column character (y_char)

    # Convert column character to numeric index (0 for 'A', 1 for 'B' and etc.)
    y = ord(y_char.upper()) - ord('A')

    # Check if the target intersection is unoccupied
    if not is_occupied(board, x, y):
        return False  # Return False if the intersection is occupied or out of bou

    # Place the stone on the board
    board[x][y] = stoneY
    return True  # Return True to indicate successful placement

# Example usage
size = 9
board = create_board(size)  # Create a 9x9 game board
position_to_place = (0, 'A')  # Row index 0, Column 'A'
stone_to_place = "●"  # Stone symbol
success = place_on_board(board, stone_to_place, position_to_place)  # Try to place
print(success)  # Print whether the placement was successful or not
```

```
False
```

In [10]:

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

#We need to include the two other def function that we created previously(ie. crea
#def create_board(size):
    #return [[" " for _ in range(size)] for _ in range(size)]

#def is_occupied(board, x, y):
    #size = len(board)
    #if 0 <= x < size and 0 <= y < size:
        #return board[x][y] != " "
    #return False


# First Define a function to place a stone on the board which has three parameter
#def place_on_board(board, stone, position):
    #x, y_char = position  # Extract row index (x) and column character (y_char)

    # Convert column character to numeric index (0 for 'A', 1 for 'B' and etc.)
    #y = ord(y_char.upper()) - ord('A')

    # Check if the target intersection is unoccupied
    #if not is_occupied(board, x, y):
        #return False  # Return False if the intersection is occupied or out of bo

    # Place the stone on the board
    #board[x][y] = stone
    #return True  # Return True to indicate successful placement

# Example usage
#size = 9
#board = create_board(size)  # Create a 9x9 game board
#position_to_place = (0, 'A')  # Row index 0, Column 'A'
#stone_to_place = "●"  # Stone symbol
#success = place_on_board(board, stone_to_place, position_to_place)  # Try to plac
#print(success)  # Print whether the placement was successful or not
```

3.5 Printing the Board

In [11]:

```
#define the print_board function using the parameter board
# next use the size function to determine the length of the board

def print_board(board):
    size = len(board)

    # Print column indices
    print("    ", end="")
    for i in range(size):
        print(chr(65 + i), end="  ")
    print()

    # Print top border
    print("   ", end="")
    for _ in range(size):
        print("--", end="")
    print("-")

    # Print rows with grid, indices, and stones
    for i in range(size):
        print(f"{i:d} |", end="")
        for j in range(size):
            print(board[i][j], end="|")
        print()

        # Print row separator which makes the board row separator
        print("   ", end="")
        for _ in range(size):
            print("--", end="")
        print("-")

#Testing the def function using the print_board(board)
size = 9
board = create_board(size)
success = place_on_board(board, "●", (2, 'A'))  # Placing a stone at intersection
success = place_on_board(board, "○", (5, 'B'))  # Placing a stone at intersection
print_board(board)
```

```
    A B C D E F G H I
   -------------------
0 | | | | | | | | | |
   -------------------
1 | | | | | | | | | |
   -------------------
2 | | | | | | | | | |
   -------------------
3 | | | | | | | | | |
   -------------------
4 | | | | | | | | | |
   -------------------
5 | | | | | | | | | |
   -------------------
6 | | | | | | | | | |
   -------------------
7 | | | | | | | | | |
   -------------------
8 | | | | | | | | | |
   -------------------
```

In [12]:

```
# Test code for 3.5 here [The code in this cell should be commented]
#define the print_board function using the parameter board
# next use the size function to determine the length of the board

#def print_board(board):
    #size = len(board)

    # Print column indices
    #print("   ", end="")
    #for i in range(size):
        #print(chr(65 + i), end=" ")
    #print()

    # Print top border
    #print("  ", end="")
    #for _ in range(size):
        #print("--", end="")
    #print("-")

    # Print rows with grid, indices, and stones
    #for i in range(size):
        #print(f"{i:d} |", end="")
        #for j in range(size):
            #print(board[i][j], end="|")
        #print()

        # Print row separator which makes the board row separator
        #print("  ", end="")
        #for _ in range(size):
            #print("--", end="")
        #print("-")

#Testing the def function using the print_board(board)
#size = 9
#board = create_board(size)
#success = place_on_board(board, "●", (2, 'A'))  # Placing a stone at intersection
#success = place_on_board(board, "○", (5, 'B'))  # Placing a stone at intersection
#print_board(board)
```

3.6 Check Available Moves

In [13]:

```
# Define a function to check available moves on the board
def check_available_moves(board):
    size = len(board)  # Get the size of the board
    moves = []  # Initialize an empty list to store available moves

    # Iterate through each row and column index on the board
    for i in range(size):
        for j in range(size):
            # Check if the intersection is not occupied
            if not is_occupied(board, i, j):
                moves.append((i, j))  # Append the unoccupied intersection to the

    return moves  # Return the list of available moves
# Print available moves
available_moves = check_available_moves(board)
print(available_moves)
```

```
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0,
8), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7),
(1, 8), (2, 0), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (2, 6), (2,
7), (2, 8), (3, 0), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (3, 6),
(3, 7), (3, 8), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4), (4, 5), (4,
6), (4, 7), (4, 8), (5, 0), (5, 1), (5, 2), (5, 3), (5, 4), (5, 5),
(5, 6), (5, 7), (5, 8), (6, 0), (6, 1), (6, 2), (6, 3), (6, 4), (6,
5), (6, 6), (6, 7), (6, 8), (7, 0), (7, 1), (7, 2), (7, 3), (7, 4),
(7, 5), (7, 6), (7, 7), (7, 8), (8, 0), (8, 1), (8, 2), (8, 3), (8,
4), (8, 5), (8, 6), (8, 7), (8, 8)]
```

In [14]:

```
# Test code for 3.6 here [The code in this cell should be commented]
# Define a function to check available moves on the board
#def check_available_moves(board):
    #size = len(board)  # Get the size of the board
    #moves = []  # Initialize an empty list to store available moves

    # Iterate through each row and column index on the board
    #for i in range(size):
        #for j in range(size):
            # Check if the intersection is not occupied
            #if not is_occupied(board, i, j):
                #moves.append((i, j))  # Append the unoccupied intersection to the

    #return moves  # Return the list of available moves
# Print available moves
#available_moves = check_available_moves(board)
#print (available_moves)
```

3.7 Check for the Winner

In [23]:

```
#Implement code for 3.7 here
#Define the function check_for_winner their parameter is board

def check_for_winner(board):
    size = len(board)

    # Iterate through each cell on the board
    for row in range(size):
        for col in range(size):
            position = (row, col)

            # Check if the cell is occupied by a stone
            if is_occupied(board, row, col):
                stone_color = board[row][col]

                # Check horizontal, vertical, and diagonal lines
                for dx, dy in [(1, 0), (0, 1), (1, 1), (1, -1)]:
                    x, y = position

                    # Check for a potential winning sequence of stones
                    for _ in range(5):
                        # Check if the current cell is out of bounds or doesn't ha
                        if not is_valid_position(board, x, y) or board[x][y] != st
                            break
                        x += dx
                        y += dy
                    else:
                        # This will run if the loop completes without a 'break'
                        # Return the color of the stone that forms a winning seque
                        return stone_color

    # Check for a draw or still available moves
    available_moves = check_available_moves(board)

    # If there are no available moves, the game is a draw
    if len(available_moves) == 0:
        return "Draw"
    else:
        # If there are still available moves, there is no winner yet
        return None

# Example usage for a 9 by 9 board:
size = 9
board = create_board(size)

# Checking the winner of the game
winner = check_for_winner(board)
if winner:
    print(f"Winner: {winner}")
else:
    print("No winner yet.")
```

```
No winner yet.
```

In [16]:

```
# Test code for 3.7 here [The code in this cell should be commented
#Implement code for 3.7 here
#Define the function check_for_winner their parameter is board
#def check_for_winner(board):

    #size = len(board)

    # Iterate through each cell on the board
    #for row in range(size):
        #for col in range(size):
            #position = (row, col)

            # Check if the cell is occupied by a stone
            #if is_occupied(board, row, col):
                #stone_color = board[row][col]

                # Check horizontal, vertical, and diagonal lines
                #for dx, dy in [(1, 0), (0, 1), (1, 1), (1, -1)]:
                    #x, y = position

                    # Check for a potential winning sequence of stones
                    #for _ in range(5):
                        # Check if the current cell is out of bounds or doesn't ha
                        #if not is_valid_position(board, x, y) or board[x][y] != s
                            #break
                        #x += dx
                        #y += dy
                    #else:
                        # This will run if the loop completes without a 'break'
                        # Return the color of the stone that forms a winning seque
                        #return stone_color

    # Check for a draw or still available moves
    #available_moves = check_available_moves(board)

    # If there are no available moves, the game is a draw
    #if len(available_moves) == 0:
        #return "Draw"
    #else:
        # If there are still available moves, there is no winner yet
        #return None

# Example usage for a 9 by 9 board:
#size = 9
#board = create_board(size)

# Checking the winner of the game
#winner = check_for_winner(board)
#if winner:
    #print(f"Winner: {winner}")
#else:
    #print("No winner yet.")
```

```
No winner yet.
```

3.8 Random Computer Player

In [17]:

```python
# Implement code for 3.8 here
import random

def random_computer_player(board, player_move):
    size = len(board)
    player_x, player_y = player_move[0], ord(player_move[1]) - ord('A')

    valid_moves = check_available_moves(board)

    # If no valid moves are available, return None
    if len(valid_moves) == 0:
        return None

    # Choose a random valid move from all available moves
    computer_next_move = random.choice(valid_moves)
    computer_x, computer_y = computer_next_move

    # Convert the computer's move to letter-coordinate format
    computer_move = (computer_x, chr(computer_y + ord('A')))

    return computer_move

# Testing the code
size = 9
board = create_board(size)

# Define the player's move in letter-coordinate format
player_move = (4, 'E')

# Getting the computer's next move
computer_next_move = random_computer_player(board, player_move)

if computer_next_move is not None:
    print(f"Computer's next move: {computer_next_move[0]}{computer_next_move[1]}")
else:
    print("No valid moves for the computer.")
```

```
Computer's next move: 4F
```

In [18]:

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

#import random

#def random_computer_player(board, player_move):
    #size = len(board)
    #player_x, player_y = player_move[0], ord(player_move[1]) - ord('A')

    #valid_moves = check_available_moves(board)

    # If no valid moves are available, return None
    #if len(valid_moves) == 0:
        #return None

    # Choose a random valid move from all available moves
    #computer_next_move = random.choice(valid_moves)
    #computer_x, computer_y = computer_next_move

    # Convert the computer's move to letter-coordinate format
    #computer_move = (computer_x, chr(computer_y + ord('A')))

    #return computer_move

# Testing the code
#size = 9
#board = create_board(size)

# Define the player's move in letter-coordinate format
#player_move = (4, 'E')

# Getting the computer's next move
#computer_next_move = random_computer_player(board, player_move)

#if computer_next_move is not None:
    #print(f"Computer's next move: {computer_next_move[0]}{computer_next_move[1]}"
#else:
    #print("No valid moves for the computer.")
```

3.9 Play Game

In [*]:

```
import random

# Implement the functions you've provided previously here

def play_game():
    board = None  # Initialize the board as None
    player_positions = {"black": [], "white": []}

    while True:
        # Display the game menu options
        print("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")

        option = input("Enter your choice (1-5): ")

        if option == '1':
            # Ask for board size and mode from the user
            board_size = int(input("Enter board size (9, 13, 15): "))
            while board_size not in [9, 13, 15]:
                board_size = int(input("Invalid size. Enter board size (9, 13, 15)

            mode = input("Enter mode (PvP or PvC): ").lower()
            while mode not in ["pvp", "pvc"]:
                mode = input("Invalid mode. Enter mode (PvP or PvC): ").lower()

            # Create the board based on user input size
            board = create_board(board_size)
            player_positions = {"black": [], "white": []}  # Reset player position

            print("Game started!")

        elif option == '2':
            if board is None:
                print("No game in progress. Start a game first.")
            else:
                print_board(board)

        elif option == '3':
            if board is None:
                print("No game in progress. Start a game first.")
            else:
                if mode == "pvp":
                    player = input("Enter player (Black or White): ").lower()
                    while player not in ["black", "white"]:
                        player = input("Invalid player. Enter player (Black or Whi

                    row, col = input("Enter position (row column): ").split()
                    row = int(row)
                    col = col.upper()  # Convert column character to uppercase

                    if player == "black":
                        stone = "●"
                    else:
                        stone = "○"
```

```
                success = place_on_board(board, stone, (row, col))

                if success:
                    player_positions[player].append((row, col))
                    print_board(board)
                else:
                    print("Invalid move. The position is occupied or out of bo

            elif mode == "pvc":
                # Player's move
                player_row, player_col = input("Enter your move (row column):
                player_row = int(player_row)
                player_col = player_col.upper()  # Convert column character to
                player_stone = "●"

                success = place_on_board(board, player_stone, (player_row, pla

                if not success:
                    print("Invalid move. The position is occupied or out of bo
                    continue

                player_positions["black"].append((player_row, player_col))
                print_board(board)

                # Check for winner or draw after player's move
                winner = check_for_winner(board)
                if winner:
                    print(f"Winner: {winner}")
                    board = None
                    continue

                # Computer's move
                computer_stone = "○"
                computer_move = random_computer_player(board, (player_row, pla

                if computer_move is None:
                    print("No valid moves for the computer.")
                    continue

                computer_row, computer_col = computer_move
                place_on_board(board, computer_stone, computer_move)
                player_positions["white"].append((computer_row, computer_col))
                print_board(board)

                # Check for winner or draw after computer's move
                winner = check_for_winner(board)
                if winner:
                    print(f"Winner: {winner}")
                    board = None

        elif option == '4':
            board = None
            player_positions = {"black": [], "white": []}  # Reset player position
            print("Game reset.")

        elif option == '5':
            print("Exiting the program.")
            break

        else:
            print("Invalid option. Please select a number from 1 to 5.")
```

```
# Start the game loop
play_game()
```

```
Game Menu:
1. Start a game
2. Print the board
3. Place a stone
4. Reset the game
5. Exit

Enter your choice (1-5):
```

In [ ]:

```
# Test code for 3.9 here [The code in this cell should be commented]
#import random

# Implement the functions you've provided previously here

#def play_game():
    #board = None  # Initialize the board as None
    #player_positions = {"black": [], "white": []}

    #while True:
        # Display the game menu options
        #print("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")

        #option = input("Enter your choice (1-5): ")

        #if option == '1':
            # Ask for board size and mode from the user
            #board_size = int(input("Enter board size (9, 13, 15): "))
            #while board_size not in [9, 13, 15]:
                #board_size = int(input("Invalid size. Enter board size (9, 13, 15

            #mode = input("Enter mode (PvP or PvC): ").lower()
            #while mode not in ["pvp", "pvc"]:
                #mode = input("Invalid mode. Enter mode (PvP or PvC): ").lower()

            # Create the board based on user input size
            #board = create_board(board_size)
            #player_positions = {"black": [], "white": []}  # Reset player positio

            #print("Game started!")

        #elif option == '2':
            #if board is None:
                #print("No game in progress. Start a game first.")
            #else:
                #print_board(board)

        #elif option == '3':
            #if board is None:
                #print("No game in progress. Start a game first.")
            #else:
                #if mode == "pvp":
                    #player = input("Enter player (Black or White): ").lower()
                    #while player not in ["black", "white"]:
                        #player = input("Invalid player. Enter player (Black or Wh

                    #row, col = input("Enter position (row column): ").split()
                    #row = int(row)
                    #col = col.upper()  # Convert column character to uppercase

                    #if player == "black":
                        #stone = "●"
                    #else:
                        #stone = "○"
```

```
                #success = place_on_board(board, stone, (row, col))

                #if success:
                    #player_positions[player].append((row, col))
                    #print_board(board)
                #else:
                    #print("Invalid move. The position is occupied or out of b

            #elif mode == "pvc":
                # Player's move
                #player_row, player_col = input("Enter your move (row column):
                #player_row = int(player_row)
                #player_col = player_col.upper()  # Convert column character t
                #player_stone = "●"

                #success = place_on_board(board, player_stone, (player_row, pl

                #if not success:
                    #print("Invalid move. The position is occupied or out of b


                #player_positions["black"].append((player_row, player_col))
                #print_board(board)

                # Check for winner or draw after player's move
                #winner = check_for_winner(board)
                #if winner:
                    #print(f"Winner: {winner}")
                    #board = None


                # Computer's move
                #computer_stone = "○"
                #computer_move = random_computer_player(board, (player_row, pl

                #if computer_move is None:
                    #print("No valid moves for the computer.")


                #computer_row, computer_col = computer_move
                #place_on_board(board, computer_stone, computer_move)
                #player_positions["white"].append((computer_row, computer_col)
                #print_board(board)

                # Check for winner or draw after computer's move
                #winner = check_for_winner(board)
                #if winner:
                    #print(f"Winner: {winner}")
                    #board = None

        #elif option == '4':
            #board = None
            #player_positions = {"black": [], "white": []}  # Reset player positic
            #print("Game reset.")

        #elif option == '5':
            #print("Exiting the program.")
            #break

        #else:
```

```
            #print("Invalid option. Please select a number from 1 to 5.")

# Start the game loop
#play_game()
```

In [ ]:

```
#Run the game (Your tutor will run this cell to start playing the game)
```

Reference

1. Python Software Foundation. Python Documentation. Python.org. Retrieved from https://docs.python.org/3/
2. GeeksforGeeks. Article Title. GeeksforGeeks. Retrieved from https://www.geeksforgeeks.org/
3. Stack Overflow. Question or Answer Title. Stack Overflow. Retrieved from https://stackoverflow.com/
4. Real Python. Article Title. Real Python. Retrieved from https://realpython.com/

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.

--- End of Assignment 1 ---