AI lab Test

Name: Dharamraj Bhatt
Reg.NO: 1947216

Exercise: (1) Write a program to implement a Tic-Tac-Toe game problem using random numbers.

according to the question the game should be played automatically played by the program by generating the random numbers for players.

In this program Instead of asking the user to place a mark on the board, code finds a spot on the board at random and places the mark there. Unless a player wins, the board will be displayed after each move. If the game ends in a tie, the result is -1 otherwise it will display the winner with their random number.

the main function play_game() performs following tasks :

1. Calls create_board() to create board and initializes with 0.
2. For each player (randomly generated), calls the random_place() function to randomly choose a location on board and mark that location with the player number, alternatively.
3. Print the board after each move.
4. Evaluate the board after each move to check whether a row or column or a diagonal has the same player number. If so, displays the winner's name. If after 9 moves, there is no winners then displays -1.

Input : random number for both players.

Output:
```
[[0 0 0]
 [0 0 0]
 [0 0 0]]
Board after 1 move
[[ 0  0  0]
 [ 0  0  0]
 [21  0  0]]
Board after 2 move
[[ 0  0  0]
 [ 0  0 76]
 [21  0  0]]
Board after 3 move
[[ 0  0  0]
 [ 0  0 76]
 [21  0 21]]
Board after 4 move
[[ 0  0 76]
 [ 0  0 76]
 [21  0 21]]
Board after 5 move
[[21  0 76]
 [ 0  0 76]
 [21  0 21]]
Board after 6 move
[[21  0 76]
 [76  0 76]
 [21  0 21]]
Board after 7 move
[[21  0 76]
 [76  0 76]
 [21 21 21]]
Winner is: 21
```

program

```python
# Tic-Tac-Toe Program using
# random number in Python

# importing all necessary libraries
import numpy as np
import random
```

```
from time import sleep

# Creates an empty board
def create_board():
  return(np.array([[0, 0, 0],
          [0, 0, 0],
          [0, 0, 0]]))

# Check for empty places on board
def possibilities(board):
  l = []

  for i in range(len(board)):
    for j in range(len(board)):

      if board[i][j] == 0:
        l.append((i, j))
  return(l)

# Select a random place for the player
def random_place(board, player):
  selection = possibilities(board)
  current_loc = random.choice(selection)
  board[current_loc] = player
  return(board)

# Checks whether the player has three
# of their marks in a horizontal row
def row_win(board, player):
  for x in range(len(board)):
    win = True

    for y in range(len(board)):
      if board[x, y] != player:
        win = False
        continue

    if win == True:
      return(win)
  return(win)
```

```python
# Checks whether the player has three
# of their marks in a vertical row
def col_win(board, player):
  for x in range(len(board)):
    win = True

    for y in range(len(board)):
      if board[y][x] != player:
        win = False
        continue

    if win == True:
      return(win)
  return(win)

# Checks whether the player has three
# of their marks in a diagonal row
def diag_win(board, player):
  win = True
  y = 0
  for x in range(len(board)):
    if board[x, x] != player:
      win = False
  if win:
    return win
  win = True
  if win:
    for x in range(len(board)):
      y = len(board) - 1 - x
      if board[x, y] != player:
        win = False
  return win

# Evaluates whether there is
# a winner or a tie
def evaluate(board):
  winner = 0

  for player in [randomlist[0], randomlist[-1]]:
    if (row_win(board, player) or
      col_win(board,player) or
      diag_win(board,player)):

      winner = player

  if np.all(board != 0) and winner == 0:
    winner = -1
  return winner

# Main function to start the game
def play_game():
  board, winner, counter = create_board(), 0, 1
  print(board)
  #randomlist = random.sample(range(1, 100), 2)
  sleep(2)

  while winner == 0:
    for player in [randomlist[0], randomlist[-1]]:
      board = random_place(board, player)
      print("Board after " + str(counter) + " move")
      print(board)
      sleep(2)
      counter += 1
      winner = evaluate(board)
      if winner != 0:
        break
  return(winner)

# Driver Code
randomlist = random.sample(range(1, 100), 2)
print("Winner is: " + str(play_game()))
```

Exercise: (2) Write a program to implement the 8-Queen Problem with a heuristic function using any local search algorithm. Suppose we generalize the eight-queens problem to the N-queens problem, where the task is to place N queens on an N by N chessboard. How must the programs be changed? It is clear that there are values for N for which no solution exists (consider N=2 or N=3, for example). What happens when your program is executed for these values? How might you produce more meaningful output?

Solution :

The idea behind the n queen is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false.

Input: number of queen

Output:

```
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
True
```

code:

```
N = 8

""" A utility function to prsolution """
def printSolution(board):
  for i in range(N):
    for j in range(N):
      print(board[i][j], end = " ")
    print()

""" A Optimized function to check if
a queen can be placed on board[row][col] """
def isSafe(row, col, slashCode, backslashCode,
    rowLookup, slashCodeLookup,
          backslashCodeLookup):
  if (slashCodeLookup[slashCode[row][col]] or
    backslashCodeLookup[backslashCode[row][col]] or
    rowLookup[row]):
    return False
  return True


def solveNQueensUtil(board, col, slashCode, backslashCode,
          rowLookup, slashCodeLookup,
          backslashCodeLookup):

  if(col >= N):
    return True
  for i in range(N):
    if(isSafe(i, col, slashCode, backslashCode,
        rowLookup, slashCodeLookup,
        backslashCodeLookup)):

      """ Place this queen in board[i][col] """
      board[i][col] = 1
      rowLookup[i] = True
      slashCodeLookup[slashCode[i][col]] = True
      backslashCodeLookup[backslashCode[i][col]] = True

      """ recur to place rest of the queens """
      if(solveNQueensUtil(board, col + 1,
              slashCode, backslashCode,
              rowLookup, slashCodeLookup,
              backslashCodeLookup)):
        return True

      board[i][col] = 0
```

```
        rowLookup[i] = False
        slashCodeLookup[slashCode[i][col]] = False
        backslashCodeLookup[backslashCode[i][col]] = False


    return False

  def solveNQueens():
    board = [[0 for i in range(N)]
          for j in range(N)]

    # helper matrices
    slashCode = [[0 for i in range(N)]
            for j in range(N)]
    backslashCode = [[0 for i in range(N)]
              for j in range(N)]

    rowLookup = [False] * N


    x = 2 * N - 1
    slashCodeLookup = [False] * x
    backslashCodeLookup = [False] * x

    # initialize helper matrices
    for rr in range(N):
      for cc in range(N):
        slashCode[rr][cc] = rr + cc
        backslashCode[rr][cc] = rr - cc + 7

    if(solveNQueensUtil(board, 0, slashCode, backslashCode,
              rowLookup, slashCodeLookup,
              backslashCodeLookup) == False):
      print("Solution does not exist")
      return False

    # solution found

    printSolution(board)
    return True
```

```
# Driver Cde
solveNQueens()
```