INTRODUCTION

# This document presents a Python program designed to implement a basic two-player Tic-Tac-Toe game. The game allows two players to take turns marking positions on a 3x3 grid until one of them wins or the game ends in a draw. The code demonstrates essential programming concepts such as loops, conditionals, list operations, and function design, making it a great example for beginners learning Python.

The program includes the following key features:

1. Game Board Representation:
   * The board is represented as a list of 9 elements, where each element corresponds to a position on the grid.
   * The current state of the board is displayed after each move.
2. Player Turns:
   * Two players, Player 1 ('X') and Player 2 ('O'), alternate turns to input their chosen positions.
   * Input validation ensures players can only choose valid and unoccupied positions.
3. Win Conditions:
   * The program checks for winning combinations after each move, based on predefined conditions.
4. Draw Scenario:
   * If all positions are filled and no player has won, the game declares a draw.
5. Replay Option:
   * Players can choose to play another round after the game ends.

METHODOLOGY USED

## Here’s a breakdown of the methodology used:

1. \*\*Display Board\*\*:

The code begins by displaying the current state of the game board. This is done using print statements that format the output to show the positions of the players' markers.

2. \*\*Player Initialization\*\*:

Two players are defined with distinct symbols ('X' for Player 1 and 'O' for Player 2). This establishes the identity of each player in the game.

3. \*\*Winning Conditions\*\*:

A function named check\_conditions is defined to evaluate whether a player has won the game. It checks all possible winning combinations (rows, columns, and diagonals) to see if any of them are filled with the same player's marker.

4. \*\*Game Loop\*\*:

The startgame function encapsulates the main game loop:

- It first displays the initial board. - It then enters a loop where Player 1 and Player 2 take turns to input their desired positions on the board.

- For each player's turn, the code validates the input to ensure it is a number between 1 and 9 and checks if the selected position is empty.

- If the position is valid, the player's marker is placed on the board, and the board is displayed again. - After each move, the game checks if the current player has won using the check\_conditions function.

- If the board is full and no player has won, the game declares a draw.

5. \*\*Replay Option\*\*:

After the game concludes, the code prompts the players to decide if they want to play again. If they choose to do so, the board is reset, and the game restarts. input handling, game state management, and win condition checking in a loop to create an interactive game experience.

SOFTWARE REQUIREMENTS

1.\*\*Programming Language\*\* - \*\*Python\*\*:

The code is written in Python, so a Python interpreter (jupter notebook) is required to run the code.

2. \*\*Libraries\*\* - \*\*Standard Library\*\*:

The code does not require any external libraries; it only uses Python's built-in functions and data structures. However, familiarity with basic Python concepts such as lists, loops, and functions is necessary.

3. \*\*User Input Handling\*\* - \*\*Input Functionality\*\*:

The code uses the input() function to collect user input. The environment must support standard input operations, which is typical in most Python environments.

4.\*\*Output Display\*\* - \*\*Console/Terminal\*\*:

The game outputs information to the console or terminal. Therefore, the environment must support console output to display the game board and messages to the players.

5.\*\*Game Logic\*\* - \*\*Data Structures\*\*:

\*\*Control Structures\*\*: The code employs loops (while loops) and conditional statements (if statements) to control the flow of the game. Knowledge of these constructs is necessary for understanding the game logic.

6. \*\*Error Handling\*\* - \*\*Input Validation\*\*:

The code includes checks to ensure that user inputs are valid (i.e., within the range of 1-9 and that the selected position is empty). This requires an understanding of how to implement basic error handling and user feedback.

7. \*\*Game State Management\*\* - \*\*State Representation\*\*:

The game state is represented by a list (board), which must be initialized and updated correctly throughout the game. Understanding how to manage and update state in a program is crucial.

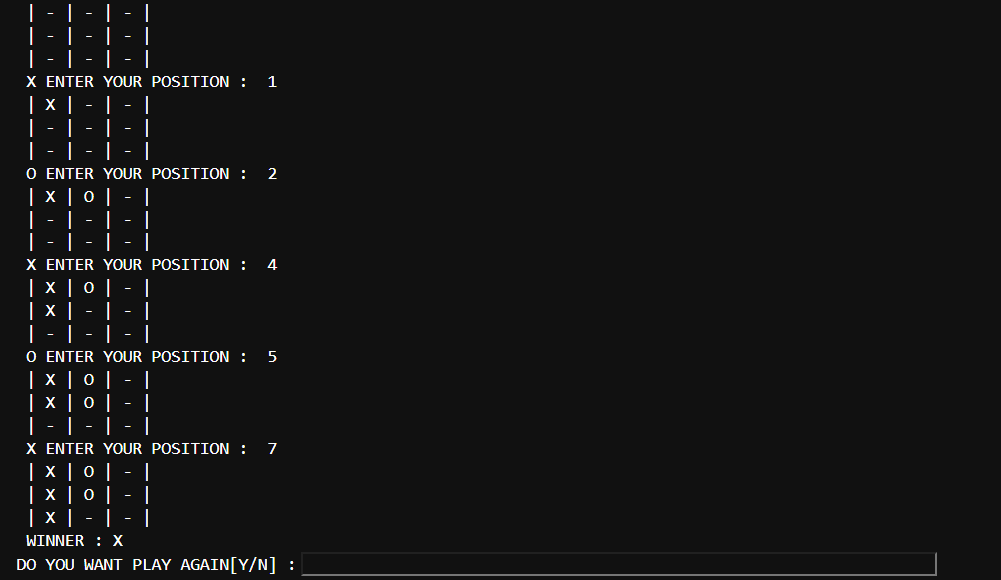
8. \*\*User Experience\*\* - \*\*User Prompts and Messages\*\*:

The game includes prompts and messages to guide the players. Clear and concise messaging is important for a good user experience.

PROGRAM CODE

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | board=["-" **for** i **in** range(9)]  **def** displayboard():  *# displaying a board in rows and columns*  print("|",board[0],"|",board[1],"|",board[2],"|")  print("|",board[3],"|",board[4],"|",board[5],"|")  print("|",board[6],"|",board[7],"|",board[8],"|")  *#creating a players*  player1="X"  player2="O"  *#wining combination and possibilites*  **def** check\_conditions(player):  conditions=[  [0,1,2],[3,4,5],[6,7,8],[0,3,6],  [1,4,7],[2,5,8],[0,4,8],[2,4,6]]  **for** check **in** conditions:  **if** board[check[0]]==player **and** board[check[1]]==player **and** board[check[2]]==player:  **return** 1  **else**:  **return** 0  *#collecting input from user*  **def** startgame():  displayboard()  **while** **True**:  **while** **True**:  player1\_option =int(input(f"{player1} ENTER YOUR POSITION : " ))  **if** player1\_option **not** **in** [i **for** i **in** range(1,10)]:  print("PLEASE ENTER A NUMBER BETWEEN [1-9]")  **continue**    **if** board[player1\_option-1]=="-":  board[player1\_option-1]=player1  displayboard()  **if** check\_conditions(player1):  **return** f'WINNER : {player1}'  **break**  **else**:  print("THIS POSITION IS NOT EMPTY")  **if** len([i **for** i **in** board **if** i=='-'])==0:  **return** 'THIS MATCH IS DRAW'  **while** **True**:  player2\_option =int(input(f"{player2} ENTER YOUR POSITION : " ))  **if** player2\_option **not** **in** [i **for** i **in** range(1,10)]:  print("PLEASE ENTER A NUMBER BETWEEN [1-9]")  **continue**    **if** board[player2\_option-1]=="-":  board[player2\_option-1]=player2  displayboard()  **if** check\_conditions(player2):  **return** f'WINNER : {player2}'  **break**  **else**:  print("THIS POSITION IS NOT EMPTY")  print(startgame())  **while** **True**:  play\_again=input("DO YOU WANT PLAY AGAIN[Y/N] :")  **if** play\_again **in** "yY":  board=["-" **for** i **in** range(9)]  print(startgame())  **else**:  exit() |

OUTPUT



FUTURE OUTCOMES AND ENHANCEMENTS

1.**Single-Player Mode**:

* Add an option for players to compete against a computer opponent.
* Implement an AI-based opponent using algorithms like Minimax to make intelligent moves.

2. **Graphical User Interface (GUI)**:

* Make the game visually appealing with clickable buttons for moves and real-time updates on the game board.

3. **Multiplayer (Online) Mode**:

* Allow players to compete over a network using Python libraries like socket or Flask for basic networking..

4. **Game Customization**:

* Allow users to customize board size (e.g., 4x4 or 5x5 grids) or create larger versions of Tic-Tac-Toe like Gomoku.

5. **Score Tracking**:

* Implement a scoring system to track wins, losses, and draws across multiple games.
* Store scores locally using a text file or database.

6. **Enhanced Input Validation**:

* Provide more robust error handling to deal with invalid inputs and edge cases.
* Add a timeout mechanism for players who take too long to input their moves.

7. **Educational Features**:

* Integrate tutorials or hints to teach players the game’s strategies, especially for beginners.
* Highlight possible winning moves or warn about threats from the opponent.

8. **Advanced AI Enhancements**:

* Improve the computer opponent’s intelligence by incorporating machine learning models that adapt to player strategies over time.

