Minior Project 1 [ Tic - Tac - Toe]

1. Introduction

1.1 Problem Statement

The problem is to create a functional, interactive Tic-Tac-Toe (XOX) game for two players. The game must allow players to take turns marking spaces on a 3x3 grid, identify a winner, and recognize a draw. The program needs to manage the game state, validate player input, and provide clear feedback to the users.

1.2 Objectives

The primary objectives are to:

Develop a two-player command-line Tic-Tac-Toe game.

Implement a clear and readable display of the game board.

Accurately determine winning conditions for rows, columns, and diagonals.

Detect when the game results in a draw.

Handle and validate player input to ensure it is within the specified range and that a chosen position is not already taken.

Manage the turn-based gameplay, switching between players 'X' and 'O'.

1.3 Scope of the Project

This project is a simple, single-file command-line application. It focuses on the core logic of the Tic-Tac-Toe game. The scope does not include graphical user interfaces (GUIs), multiplayer networking, or an AI opponent. The game is designed to be played by two human players on the same machine.

2. Technology Stack Used

2.1 Programming Languages

Python: The entire game logic and structure are written in Python.

2.2 Libraries/Frameworks

No external libraries or frameworks are used. The project relies solely on Python's built-in functionalities.

2.3 Tools and Platforms

Python Interpreter: Used to run the Python script.

Command-line Interface (CLI): The game is executed and played within a terminal or command prompt.

3. System Architecture

3.1 Architecture Diagram

+------------------+

| Game Logic |

| (play\_game()) |

+------------------+

|

| Calls

V

+------------------+ +------------------+

| Board Display | <-> | Game State |

| (print\_board()) | | (board list) |

+------------------+ +------------------+

| |

| Calls | Checks

V V

+------------------+ +------------------+

| Win Checker | <-> | Draw Checker |

| (check\_winner()) | | (is\_draw()) |

+------------------+ +------------------+

3.2 Module Description

The system is composed of several modular functions, each with a specific responsibility:

play\_game(): The main function that orchestrates the entire game. It initializes the board and players, contains the main game loop, handles player input, and manages the game flow.

print\_board(board): A utility function responsible for rendering the game board in a user-friendly, 3x3 grid format on the console.

check\_winner(board, player): A logic function that checks for a winning condition after each move. It iterates through predefined winning combinations to see if the current player has occupied all three cells in any of those combinations.

is\_draw(board): A simple logic function that checks if the game has ended in a draw. It does this by verifying if all cells on the board have been filled with 'X' or 'O'.

4. Dataset Description (if applicable)

This project does not use an external dataset. The game state is managed internally by a Python list, which serves as the data structure for the board.

5. Implementation

5.1 Code Flow Description

The play\_game() function is called to start the game.

It initializes a list board with strings "1" through "9" and sets the current\_player to 'X'.

The program enters an infinite while True loop that continues until a break statement is executed.

Inside the loop, print\_board(board) is called to display the current state of the game.

The program prompts the current\_player to enter their desired move (a number from 1 to 9).

Input Validation: The program checks if the input is a digit and is within the valid range (1-9). If not, it prints an error and uses continue to restart the loop, prompting the player again.

Position Check: The input is converted to a list index. The program checks if the chosen position is already taken (i.e., if the cell contains 'X' or 'O'). If so, it prints an error and uses continue to prompt the player again.

Make Move: If the input is valid and the position is free, the board list is updated with the current\_player's symbol.

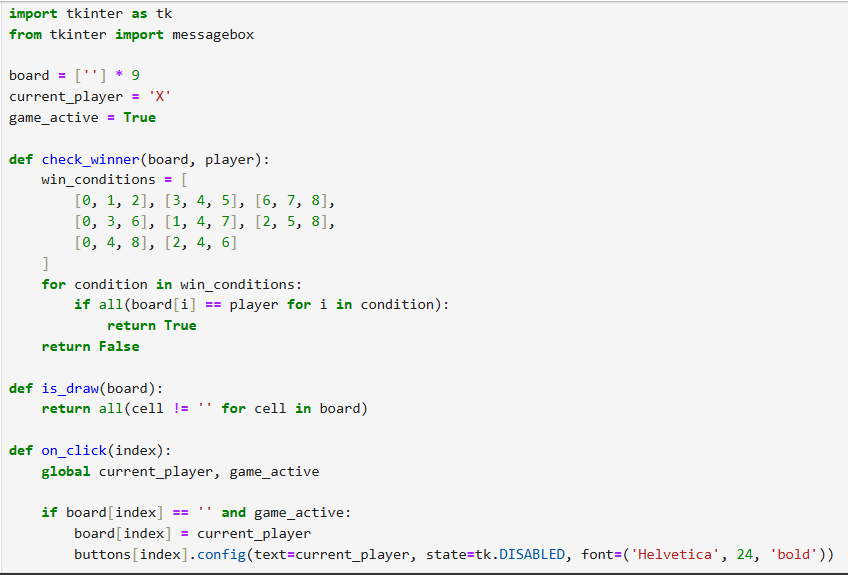
Check for Winner: The check\_winner(board, current\_player) function is called. If it returns True, the program declares the winner, prints the final board, and uses break to exit the loop.

Check for Draw: If no winner is found, the is\_draw(board) function is called. If it returns True, the game is declared a draw, the final board is printed, and break is used to exit the loop.

Switch Player: If the game has not ended, the current\_player is switched from 'X' to 'O' or vice-versa, and the loop repeats for the next turn.

The play\_game() function concludes when the while loop is broken.

5.2 Screenshots of Execution

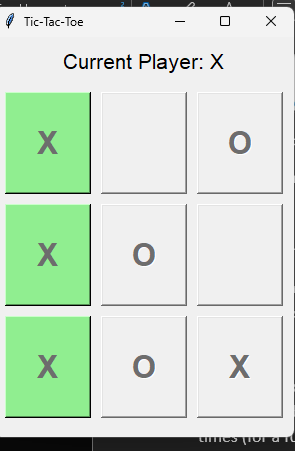






6. Results and Analysis

6.1 Output Samples



6.2 Performance Evaluation

The performance of the game is highly efficient. The logic is based on simple list manipulations and loops, which are very fast operations in Python.

Time Complexity: The check\_winner function has a constant time complexity, O(1), as it checks a fixed number of winning conditions (8 total). The main game loop runs at most 9 times (for a full board). Therefore, the overall time complexity of a single game is also constant.

Space Complexity: The space usage is minimal. The board list stores a fixed number of 9 elements. Other variables also consume a constant amount of memory. The space complexity is O(1).

7. Challenges Faced and Solutions

Challenge: Ensuring player input is valid (e.g., a number from 1-9) and that they do not overwrite a previous move.

Solution: The code implements explicit validation checks using move.isdigit() and int(move) in range(1, 10). It also checks if the target cell board[move] is already 'X' or 'O'. The use of a continue statement within the while loop efficiently handles these invalid inputs by re-prompting the user without interrupting the game's flow.

Challenge: The logic for detecting a win could be complex and repetitive if each winning combination was checked individually.

Solution: A list of win\_conditions was created. A single for loop iterates through this list, and a generator expression with all() efficiently checks each condition. This approach is clean, readable, and easily extensible if the board size were to change.

8. Conclusion

This project successfully delivers a complete and functional command-line Tic-Tac-Toe game. It demonstrates a solid understanding of fundamental programming concepts, including functions, loops, conditional statements, and data structures. The code is well-structured, easy to read, and robust enough to handle various user inputs and game scenarios. The modular design, with separate functions for displaying the board and checking for game-ending conditions, makes the code maintainable and easy to understand. This simple project serves as an excellent foundation for more complex games or for introducing game development concepts.

9. References

Python's official documentation for basic data types, control flow, and functions.