Algorithms and Programming Final Project

Ultimate Tic-Tac-Toe

Student Name: Bernard Choa

Student ID: 2502022414

Class: L1CC

Lecturer’s Name: Jude Joseph Lamug Martinez

Project Specification

To describe this project in simple terms, I have created a digital version of Ultimate Tic Tac Toe, built with the Python programming language and Tkinter module, that can be played with two players locally. This project is aimed towards anyone who is looking for a way to pass time with casual fun and thrill. While most people would typically go for chess, the game has a very difficult learning curve that might dissuade and alienate casual players. Checkers is another popular board game that is simpler than chess but also lacks its depth, as there are only so many unique ways to move the pieces one square diagonally. To me, Ultimate Tic Tac Toe strikes the perfect balance between simplicity and depth, and that is why my goal is to popularise this game, even if only by a little, so that more people may enjoy and appreciate the game, and to hopefully cement its position and status as a classic board game with the likes of chess, checkers, etc. Considering that both chess and checkers have already had their own championships, and subsequently, their own grandmasters, as such it's only natural to eventually host the first Ultimate Tic Tac Toe championship one day, and I do hope that this game I made can help us inch closer to making that dream come true.

Input

1. Mouse for selecting which tile to mark

Output

1. Tiles that are marked by the first player
2. Tiles that are marked by the second player
3. Grey tiles (Inactive)
4. White tiles (Active)
5. Tiles coloured by the colour of the first player (Indicates that the first player has achieved local victory on that mini-grid)
6. Tiles coloured by the colour of the second player (Indicates that the second player has achieved local victory on that mini-grid)
7. Announcer above the 81x81 grid that announces which player's turn and who achieved the global victory

Designing the Solution

There are three main components that are absolutely indispensable to the game; an announcer, a restart button, and the grid from which the actual game will be played it. The announcer helps to make clear what is happening and what to do at various points of a typical match, such as which player's turn,.whether there's a draw, and which player comes out as the victor. The restart button allows the players to painlessly reset the game at any point during the match, whether it's right after a player has just won decisively, or if the players agreed to redo the match due to an unforeseen glitch/error.

Implementation

The project is made up of only one file. I used the TKInter module in Python to create the window from which the game will be played. The base design of the game was inspired by a YouTube tutorial made by BroCode who used the TKInter module to build the GUI, as well as the random module to select a player randomly at the beginning of the game. My implementation of Ultimate Tic Tac Toe uses the functional paradigm. I defined the functions:

- next_turn()
- check_winner_local()
- check_winner_global()
- empty_spaces_local()
- new_game()

The programme begins by defining a Tk() object referred to as the main_window, this window will display everything necessary for the game (announcer, the interface for the game, & the restart button). The programme then initialises the variables that would represent the players and the grid.

```
# Initialises main window
main_window = Tk()
main_window.title("Buttons")
# Initialises the players
players = ["x", "o"]
player = random.choice(players)
# Initialises the grid
grid1 = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
grid2 = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
grid3 = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
grid4 = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
grid5 = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
grid6 = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
grid7 = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
grid8 = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
grid9 = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
megagrid = [[grid1, grid2, grid3], [grid4, grid5, grid6], [grid7, grid8, grid9]]
grid_status = [["", "", ""], ["", "", ""], ["", "", ""]]
act_grid = [None, None]
```

Image 1. The first part of the driver code

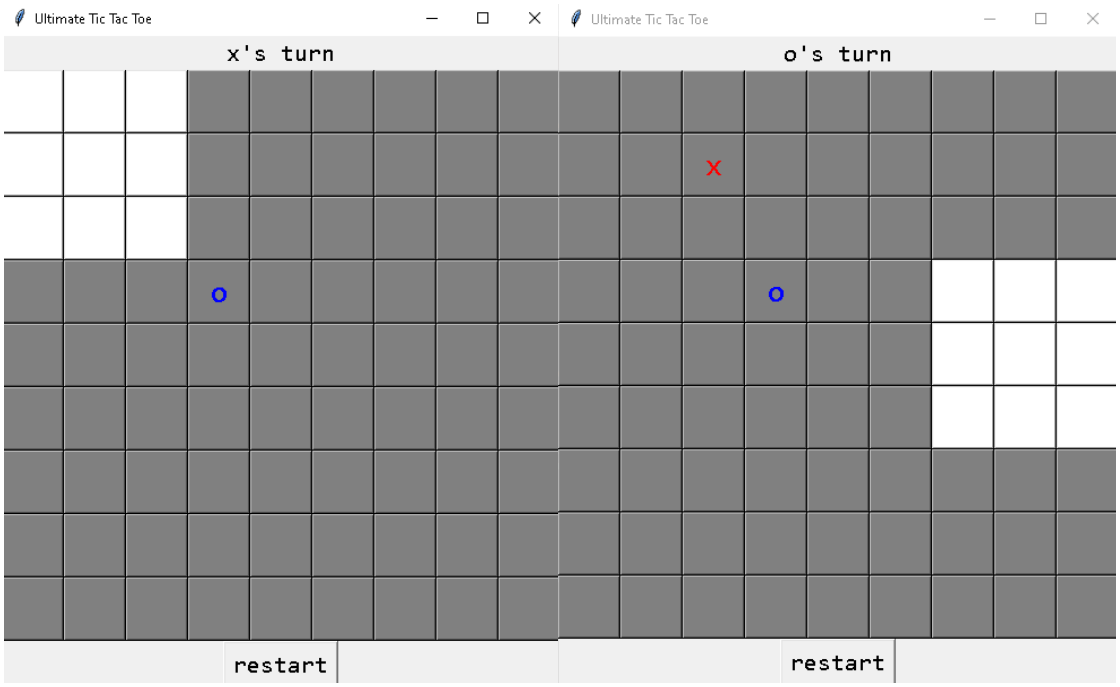

Image 2. First player (O) chose the Northwestern tile in the 5th mini-grid (middle), next player chose the Eastern tile in the 6th mini-grid (Northwestern).

The second part of the driver code is for displaying the announcer, the game itself, & the restart button in the window. The restart button triggers the new_game() function which resets all the tiles as blank, starting a brand new round.

```
act_grid = [None, None]

# Displays announcer
announcer = Label(font=("consolas", 16), text=f"{player}'s turn")
announcer.pack(side="top")

# Displays restart button
restarter = Button(font=("consolas", 16), text="restart", command=new_game)
restarter.pack(side="bottom")

# Displays grid
frame = Frame(main_window)
frame.pack()

# Initialises every integer from the megagrid nested list as a Button object
```

Image 3. Second part of the driver code

The third and final part of the driver code is dedicated to instantiating and running the game itself. It iterates through all 81 tiles and redefines them as Button() objects that triggers the next_turn() function. This function is the centerpiece of the programme, as it does everything from checking for any tiles that satisfies any of the win conditions (3 matching tiles orthogonally/diagonally), dictates what are the possible moves the next player can take, etc.

```
# Initialises every integer from the megagrid nested list as a Button object
for r_gr in range(3):
    for c_gr in range(3):
        for row in range(3):
            for col in range(3):
                megagrid[r_gr][c_gr][row][col] = Button(frame, text="", font=("consolas", 20), bg="white", width=3, height=1,
                                        command = lambda r_gr = r_gr, c_gr = c_gr, row = row, col = col: next_turn(r_gr, c_gr, row, col))
                megagrid[r_gr][c_gr][row][col].grid(row=(3 * r_gr) + row, column=(3 * c_gr) + col)


main_window.mainloop()
```

Image 4. Third and final part of the driver code

next_turn() is a very complicated function, but it mainly consists of two parts, what I would call the “main branch” and the “alternate branch”. It takes in four parameters; The main branch is fairly simple, it checks whether or not the current player achieved a local or global victory, as well as colour coding the mini-grids appropriately. The alternate branch deals with the possibility of a “dead” grid (“dead” as in either a player had won the mini-grid or it was a draw), and allows the player to play on 72 of 81 tiles.

```
# Activates every time a button is pressed
def next_turn(r_gr, c_gr, row, col):
    global player, act_grid
    # Main branch of the game
    if megagrid[r_gr][c_gr][row][col]['text'] == "" and not check_winner_global() and not grid_status[r_gr][c_gr] and not grid_status[row][col] and (act_grid == [None, None] or
        # Player x's turn
        if player == players[0]:
            megagrid[r_gr][c_gr][row][col]['text'] = player
            megagrid[r_gr][c_gr][row][col].config(fg="red")
            # Sets next turn to player o after checking that there's no winner yet
            if check_winner_global() == False:
                player = players[1]
                announcer.config(text=(players[1] + "'s turn"))
            # Declares player x as winner
            elif check_winner_global() == True:
                announcer.config(text=(players[0] + " wins"))
            # Declares tie
            elif check_winner_global() == "Tie":
                announcer.config(text=("It's a tie!"))
            # Sets next active grid
            act_grid[0] = row
            act_grid[1] = col

        # Player o's turn
```

Image 5. The main branch of next_turn() that deals with Player X's win conditions. The long if statement prevents any player from marking a tile if the tile has already been marked, if global victory has been achieved, if local victory has been achieved, if the next active grid turns to be occupied, and if a grid has been fully occupied without a victor (draw)

The next_turn function itself consists of three other functions, namely check_winner_local(), check_winner_global(), and empty_spaces_local(). check _winner_local() detects if a player scored 3 consecutive marks orthogonally/diagonally in any of the mini-grids.

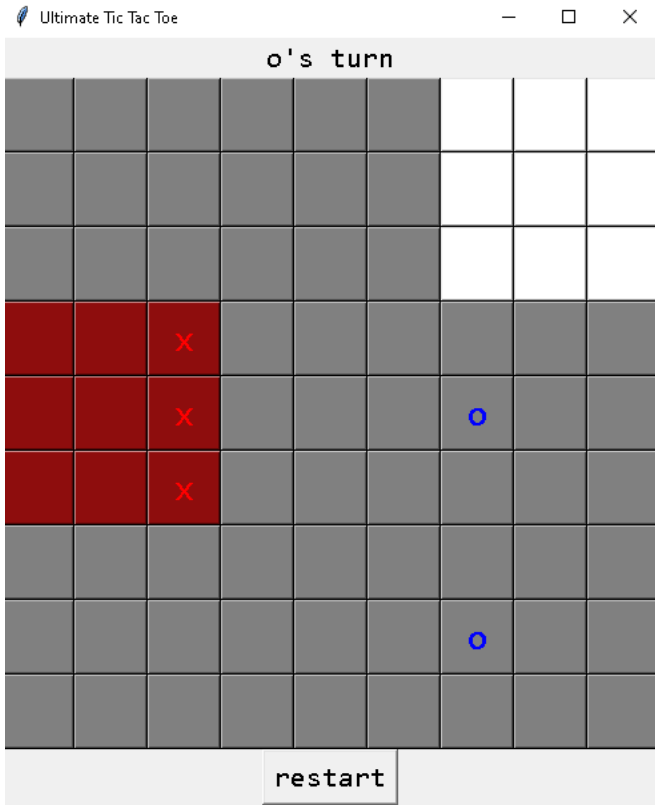

Image 6. check_winner_local() detects that Player X won locally on the Western grid

Whereas check_winner_global() detects if a player won 3 mini-grids in such a way that they line up orthogonally or diagonally.

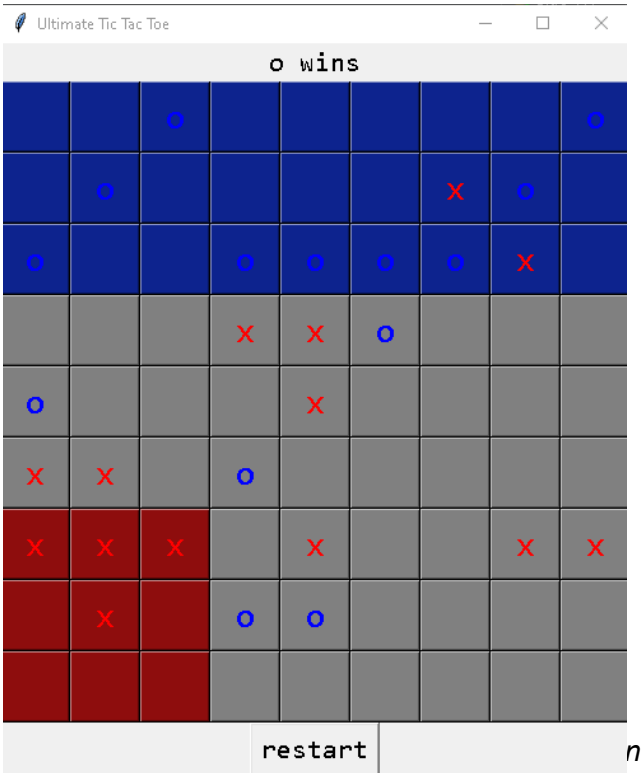

Image 7. n globally

empty_spaces_local() tracks the amount of space left in all mini-grids. All mini-grids start off with a space count of 9 at the beginning, and as players take turns to mark the tiles, the count diminishes until a mini-grid reaches 0, at which point said mini-grid will be coloured yellow to indicate a draw (locally).

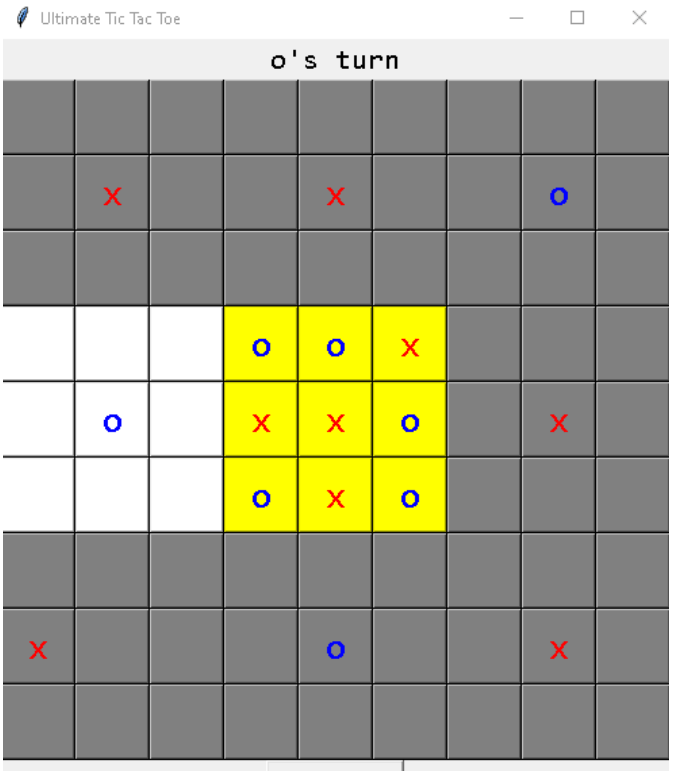

Image 8. empty_spaces_local detects that the Middle mini-grid's count reached zero, and colour coded the mini-grid as yellow

Last but not least, there is the new_game() function that only activates when the restart button is pressed.

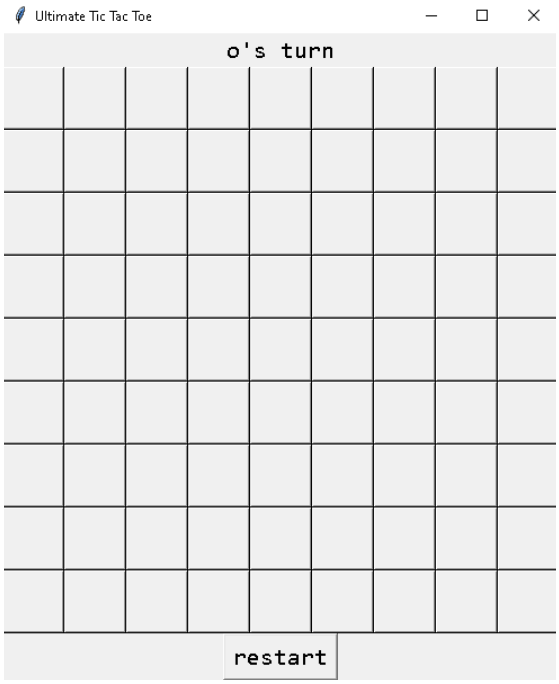

Image 9. How a fresh Ultimate Tic Tac Toe round looks like

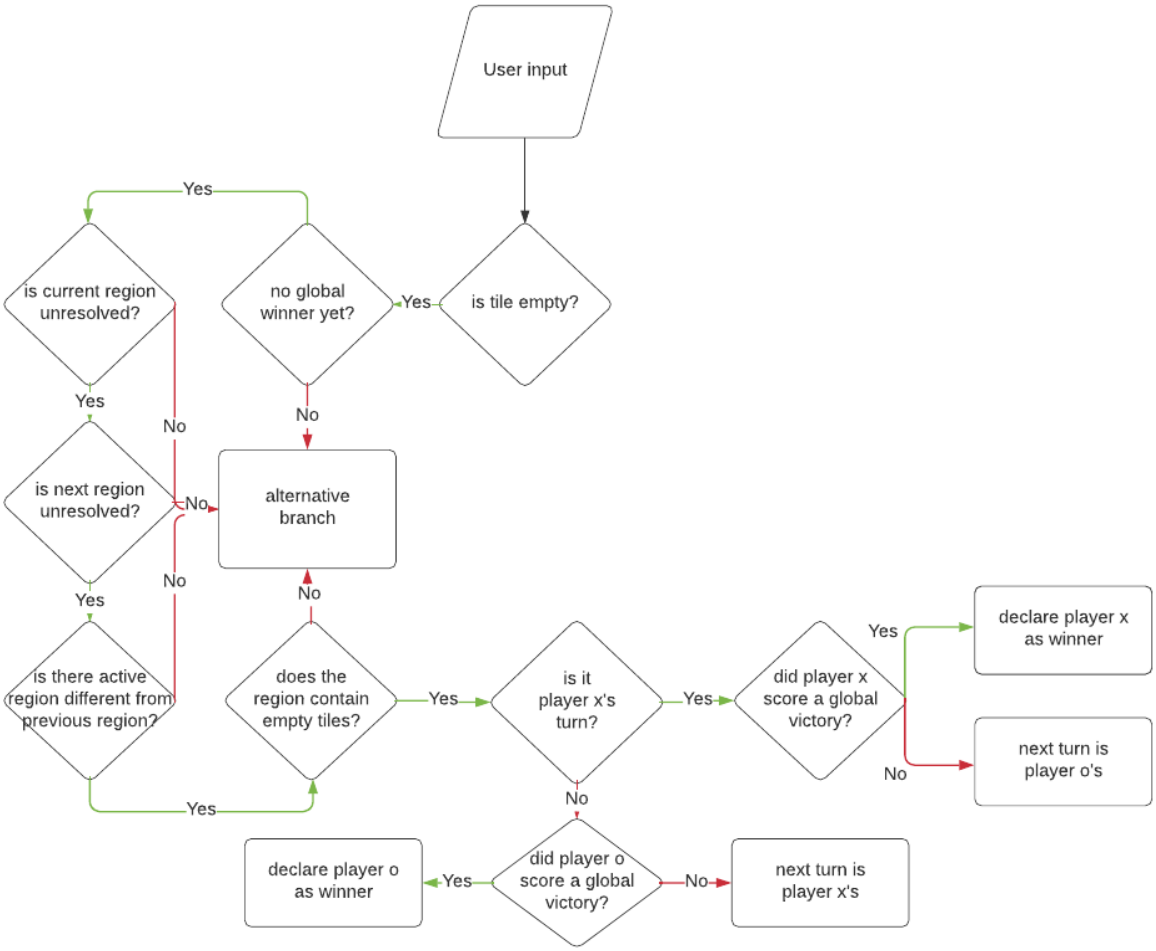

Image 10. Flowchart of the main branch

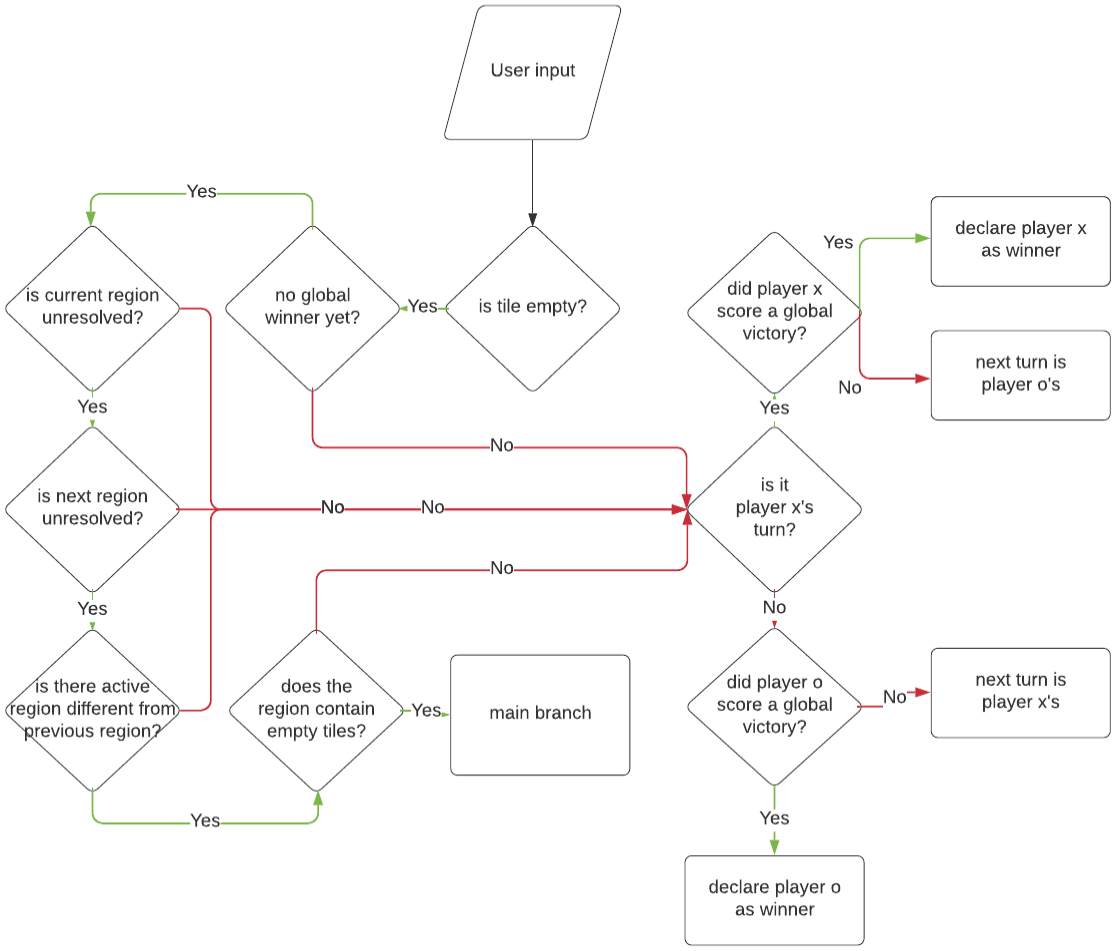

Image 11. Flowchart of the alternate branch

Reflection

Knowing that creating a working version of Ultimate Tic Tac Toe in Python would be easier if I knew how to create the basic version of Tic Tac Toe beforehand, I spent some time surfing for existing iterations of Tic Tac Toe in Python on the Internet. .Soon I found a tutorial video made by BroCode that satisfied my requirements. His implementation of Tic Tac Toe uses the tkinter module to create Button() objects that would represent the 9x9 grid.

I wanted to design Ultimate Tic Tac Toe in such a way as to enable anyone to get into the game immediately, no strings attached. However, I must take into account the player's familiarity with the game. I have seen plenty of guides on the Internet, and they do a rather underwhelming job at explaining the rules of the game, especially those rules which are concerned with the movement of the players throughout the 81x81 grid. Adding a button that redirects players to a poorly written guide that is hard to understand will only bog down the

experience. Eventually, I realised that the game is best explained and understood hands on. I added white, gray, red, and blue tiles to telegraph which tiles are playable and which are not, in hopes of intuitively teaching the rules of the game to any beginner.

I learned quite a lot from working on this project. To summarise, I now have a rough idea of how developing a programme would typically feel like. I was reminded of a saying to stand on the shoulder of giants, and that piece of wisdom helped tremendously in developing Ultimate Tic Tac Toe, as the foundation from which I built off of (BroCode's Python tutorial of Tic Tac Toe) was already secure and reliable, and that allowed me to concentrate my efforts on extending the foundational code for Ultimate Tic Tac Toe. Speaking in less general terms and more towards the technical aspects of programming, I noticed a pattern; that the while loop is an invaluable component for the game's functioning. Finally, what is most impressive about BroCode's iteration to me is the classes, or lack thereof, as the code is entirely functional, which is proof that programming a game can be done without an object-oriented paradigm.