# Tic Tac Toe Command Line Game

# CMP5361

By Aman Zamarad - 21183681

# Introduction

In this logbook I will show the various stages of creating a program, in this case a Tic Tac Toe game. Firstly, I will start off with planning and decomposing the program to make it easier to work on as well as making it easy for others to follow. I will be using Gherkins specifications, data models, and a mathematical approach, to help me understand the design of the program and allow me to map out the requirements.

After the planning stage, I will document the implementation of the program and carry out various tests to ensure the features are working correctly. The implementation of the program will be done one python as I already have previous experience on it.

Lastly, I will explore a team-based software system which will further develop my understanding on team-based project, and I will discuss how I could leverage this version control system if I was to apply it to the implementation of this Tic Tac Toe project.

**The main features I aim to implement are**:

- Allow a player to Start a new 2-player game
- Allow each player to take turns placing tokens
- Allow the game to judge when a given player(X or O) has won the game
- Allow a player to start a single player game against the CPU

# Using Gherkin Specification to describe the expected behaviour of each feature

**What is Gherkin specification and why is it necessary?**

A specification is an analysis of what is required within a system. I am using gherkin's specification for this planning stage to map out all the possible scenarios that can come from each feature from the program. This is by starting with a feature, then mapping out all the possible scenarios that can come from it. Each scenario is further defined by "given", "when" and "then" steps, showing the behaviours of the program. It also allows stakeholders, developers, and testers to collaborate and understand as it is in a human-readable format (Paredes, 2021).

## Feature: Choosing Game mode from Main Menu

This is a list of options on the command line which the CLP(Command Line Player) can pick. These options are "Start single player", "Start Two Player" and "Exit Program". The CLP must pick from one of the options, by entering one of the numbers between 1-3. A main menu is required so the user(s) that want to start a game can choose between the options.

**Here is a simple layout of how the main menu would look:**

Tic Tac Toe Command Line Game

Main Menu

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

1. Start Single Player
2. Start Two Player
3. Exit Program

Below are all the possible scenarios that can come from the Command Line Player's inputs when using the "Choose game mode from main menu" feature. By Using Gherkin specifications based on Hoare's Logic, this will provide the postconditions based on the actions on the precondition.

Scenario 1: Starting a single player game from the main menu

**Given** the main menu is displayed via standard output

**And** I have input the value “1” via command prompt

**When** I confirm my choice by pressing the Enter key

**Then** the program should display the message “You have chosen Single Player Game”

**And** the program should transfer to the “Start a single player game” feature

**And** a blank game board should be displayed via standard output

Scenario 2: Starting a two-player game from the main menu

**Given** the main menu is displayed via standard output

**And** I have input the value “2” via command prompt

**When** I confirm my choice by pressing the Enter key

**Then** the program should display the message “You have chosen Two Player Game”

**And** the program should transfer to the “Start a two player game” feature

**And** a blank game board should be displayed via standard output

Scenario 3: Exiting the program from the main menu

**Given** the main menu is displayed via standard output

**And** I have input the value “3” via command prompt

**When** I confirm my choice by pressing the Enter key

**Then** the program should display the message “Program ended abruptly”

Scenario 4: Entering a value which isn’t one of the options

**Given** the main menu is displayed via standard output

**And** I have input and invalid value via command prompt

**When** I confirm my choice by pressing the Enter key

**Then** the program should display the message “Invalid value entered, please try again”

**And** the program should transfer to the “Choosing game mode from main menu” feature

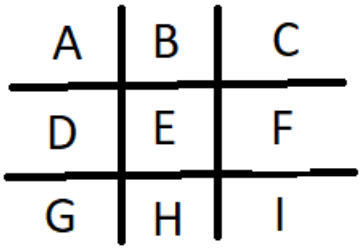
**Feature: Placing a Token on the Game Grid**

This feature is what allows the player to place a token on their specified space on the grid. This is by the player specifying the coordinate of where they want to place their token on the grid. Each space on the grid is represented with a letter coordinate, so A|B|C|D|E|F|G|H.

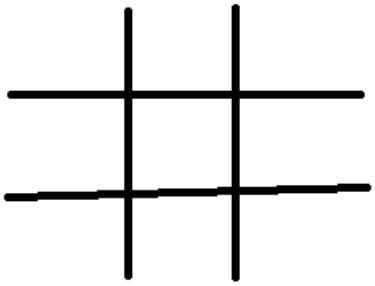
Here is a rough sketch on MS paint of how a new game grid would look when one of the game modes has been picked:

You Have Chosen Single Player

Grid with Coordinates representing each space:




New Game Board Created:



Player X Starts, pick a coordinate to place your token:

Before the main game board is created, there will be a game board displaying coordinates that represent each space in the grid. This is to help the Command line user(s) know what coordinate is representing their desired space on the grid so they can input the value via input prompt. After example board has been printed, and empty board is created, and a new game will start.

Scenario 1: Computer Placing a token on the game board

**Given** there is a single player game in play

**And** the game grid is displayed via standard output

**And** there is an available space on the game grid

**When** CPU has specified a token placement on the grid

**Then** that token should be placed within that space

**And** the updated grid should be displayed via standard output

**And** the game transfers back to “placing a token on game board” feature

Scenario 2: Real life Player Placing a token on the game board

**Given** there is a two player game in play

**And** the game grid is displayed via standard output

**When** a player has specified a token placement on the grid

**And** there is an available space(s) on the game grid

**Then** that token should be placed within that space

**And** the updated grid should be displayed via standard output

**And** the game transfers back to “placing a token on game board” feature

Scenario 3: Placing a token in an occupied space on the grid

**Given** there is a game in play

**And** the game grid is displayed via standard output

**When** a player has specified a token placement on the grid

**And** that space is occupied on the game grid

**Then** there should be a message on the standard output stating “Specified space already occupied, try again”

**And** the game transfers back to “placing a token on game board” feature

Scenario 4: Specifying an invalid coordinate

**Given** there is a game in play

**And** the game grid is displayed via standard output

**When** a player has specified a token placement that isn't on the grid

**Then** there should be a message on the standard output stating "Invalid input"

**And** the game transfers back to "placing a token on game board" feature

## Feature: Detecting If a player has won or if there is a draw

This feature is constantly in action when a game has started by detecting if a player has won the game or if both have ended the game in a draw. If a players tokens are adjacent to one another, they are the winner. However, if all spaces are taken up on the grid, then the game ends in a draw. Once the game has come to an end, the Command line player is taken back to the main menu. Without this feature, the game won't come to an end and there will be no winner announced.

Scenario 1: Player X places a game winning move

**Given** Player X has the opportunity to win the game

**And** Player X specifies a game winning token placement via input prompt

**When** Player X confirms their choice by pressing the Enter key

**Then** the program should display the message "Player X Wins!"

**And** the program should transfer to the "Choosing game mode" feature

Scenario 2: Player 0 places a game winning move

**Given** Player X has the opportunity to win the game

**And** Player X specifies a game winning token placement via input prompt

**When** Player X confirms their choice by pressing the Enter key

**And** Player 0 has 3 tokens adjacent to one another

**Then** the program should display the message "Player 0 Wins!"

**And** the program should transfer to the "Choosing game mode" feature

Scenario 3: Game ends with a draw

**Given** player X or player 0 have entered a value for their token in the last available space via command prompt

**When** Player X or player 0 presses the Enter Key

**And** there is no decided winner

**Then** the program should display the message “Game ends in a draw”

**And** the program should transfer to the “Choosing game mode” feature

Scenario 4: Game Restart

**Given** player X or player 0 have entered a value for their token in the last available space via command prompt

**When** Player X or player 0 presses the Enter Key

**And** there is no decided winner

**Then** the program should display the message “Game ends in a draw”

**And** the program should transfer to the “Replay” feature

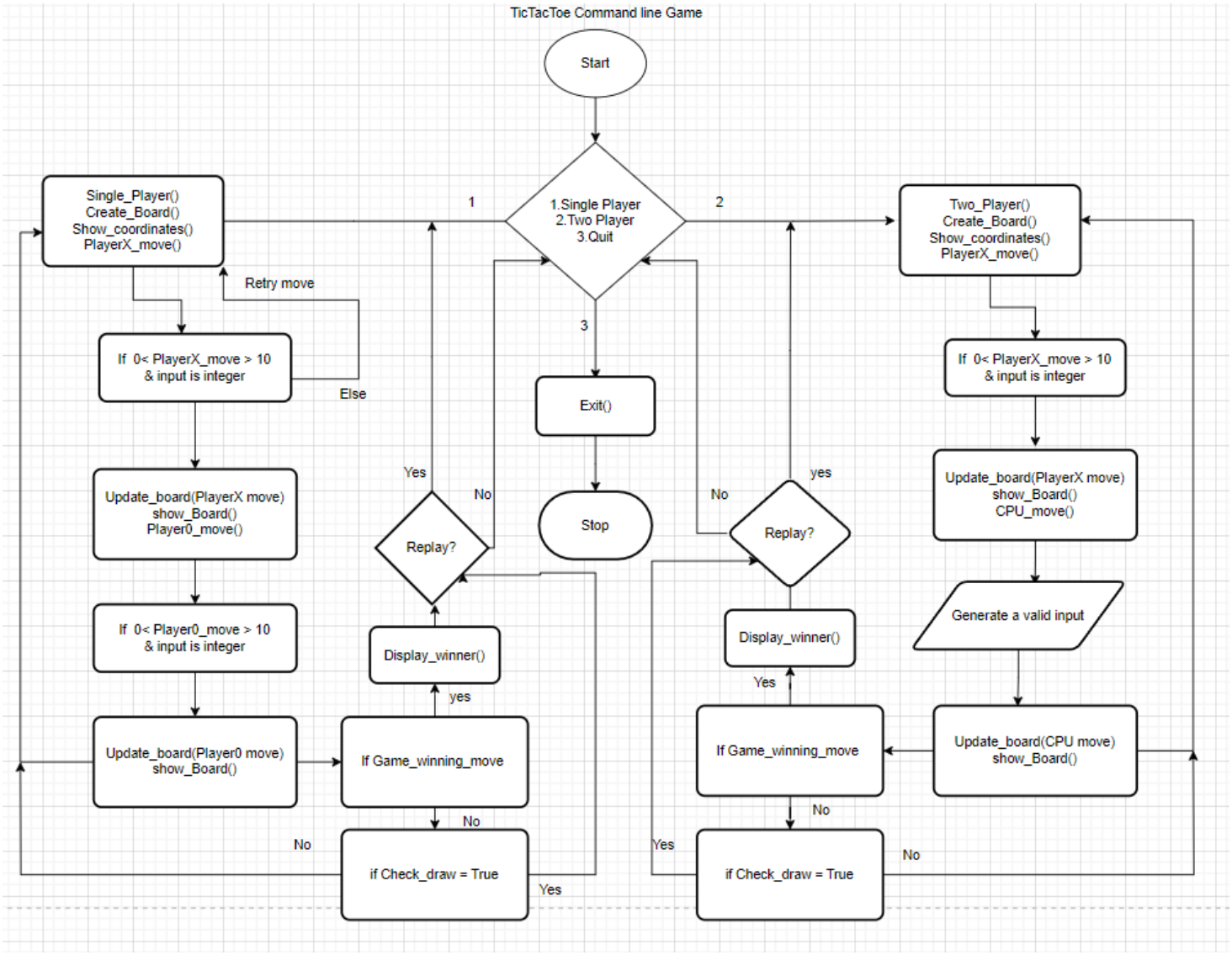
## Data modelling

Tokens: A game piece that will be placed on the grid once a player has specified their desired coordinates. The tokens can be of two possible values: X or O.

Without a token that the users can place on the grid, they won’t be able to take turns, thus the game cannot be played since they aren’t able to perform the main function. Using two distinct tokens will allow for easy player identification and strategic decision making as players can evaluate potential winning moves easier.

$$Let\ Token = \{X, O\}$$

Flowchart outlining key functions of the game:




Creating a flowchart of the overall program is beneficial in many ways. It creates a visual representation of the program's logic and flow, using various symbol to illustrate the fundamentals. It also serves as a planning tool, allowing me to break the down the projects functionality into smaller, more manageable tasks, as well as developing a systematic approach to coding. Furthermore, I can evaluate the logic of the program before I start coding it to make sure that I don't get certain conditions wrong.

## Explaining each function:

**Main_menu():**

Displays the start screen of the game, has three options:

- 1.Single player
- 2.Two player
- 3.Quit

If the user input is:

- “1”, then the Single_player() function is called
- “2”, then the Two_player() function is called
- “3”, then the Exit() function is called

$$Main\_menu: \text{void} \rightarrow \text{String}$$

This function refers to the standard output for the main menu, where the user will be welcomed .

**Single_player()**:

Starts a new single player game, calls Create_Board(), Show_coordinates() and PlayerX_move().

**Two_Player():**

Starts a new Two player game, calls Create_Board(), Show_coordinates() and PlayerX_move().

**Exit():**

Ends the program

**Create_Board():**

A new game board is created. The grid is represented by a dictionary which hold the value of each space. Since the board is being created, every space must be empty, essentially starting a new game.

For example:

Board = {A:” ”, B:” ”, C:” ”, D:” ”, E:” ”, F:” ”, G:” ”, H:” ”, I:” ”}

Or

$$Let\ Board = \{A, B, C, D, E, F, G, H, I\}$$

The Cardinality of the coordinates:

$$\#Board = 9$$

**Show_coordinates():**

This is a simple function that prints a grid with each Letter in their corresponding spaces. This is to give the Player an idea on what value they should enter if they want to specify a space for their token

The Cardinality of the coordinates can be represented as:

$$Show\_coordinates: \text{void} \rightarrow \text{String}$$

**Show_Board():**

This is the function that shows the updated Board after every turn. It will do this by accessing the Board dictionary and taking the values then printing them into a 3x3 grid that is easy for the command line user(s) to understand.

$$Show\_Board{:}Board \rightarrow String$$

**Update_Board():**

This is the function that takes the coordinates from either player X or Player O, and searches for the Key that matches the input. It then places that players token as the Value for that Key.

**PlayerX_move():**

This function is called when it is Player X's turn. It'll ask for an input that is:

- Between 1 and 9
- An integer
- Is an empty space

Once the player has entered a valid input, the Update board is called, and the input value is passed

**Player0_move():**

This function is called when it is Player O's turn. It'll ask for an input that is:

- Between 1 and 9
- An integer
- Is an empty space

Once the player has entered a valid input, the Update board is called, and the input value is passed

**CPU_move():**

Once Player X has made their move, this function is called. It checks the Board dictionary for any empty spaces and then picks one of those spaces at random. This random coordinate will be where the Token is placed for that move. The function should be coded so that:

- The random value must be within 1-9
- The value picked must be a key with an empty value
- Must be an integer

**Game_winning_move():**

This is a function that is always active in the background. After every move, it checks if the grid:

- Has three identical tokens adjacent to one another

If this condition is met, the Display_winner() function should be called.

**Check_draw():**

There will be a variable that keeps track of the amount of moves made. If the variable:

- Is equal to 9 (as there are a maximum of 9 moves you can make)

The game will be announced as a Draw and user(s) and the restart function is called

**Display_winner():**

When this function is called, the game comes to an end and the winner is announced. The restart function is called.

**Restart():**

This asks user(s) If they want to restart the game or go back to the main menu. If the user input is:

- “yes”, then the game is restarted by calling the function of the current game mode again
- “no”, then the main_menu() function is called

# Implementation Discussion

The function “printBoard” takes the value of all the keys from the “Board” Dictionary. It then places these values in human-readable format. I made it so it prints the current board out in a grid format.

The function ”display_coordinates” is a simple function that displays the coordinates of the game board so Command line players know which values to enter for when they want to place a token on the grid. I noticed later that once the function was carried out, the empty game board was printed straight after which didn’t give the user(s) time to look at the example board properly. I used to inbuilt time function to pause the execution of the program for one second so players have time to see it.

```
import time
import random
# Holds all board values
Board = {1: " ", 2: " ", 3: " ",
         4: " ", 5: " ", 6: " ",
         7: " ", 8: " ", 9: " "}

#vars
count = 0
x = 0
o = 0


# Printing the board function with values
def printBoard(board):
    print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])
    print('-----------')
    print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])
    print('-----------')
    print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])

def display_coordinates():
    print("Here is an example board with coordinates within their corresponding spaces\n")
    print("1 | 2 | 3")
    print("--|---|--")
    print("4 | 5 | 6")
    print("--|---|--")
    print("7 | 8 | 9 \n")
    print('\nCreating Empty Board...\n')
    time.sleep(1)
```

note: At first, I was using letters instead of coordinates and having each token its own custom type however I kept coming across issues when converting token into string.

This class handles all the functions that are associated with the tokens.

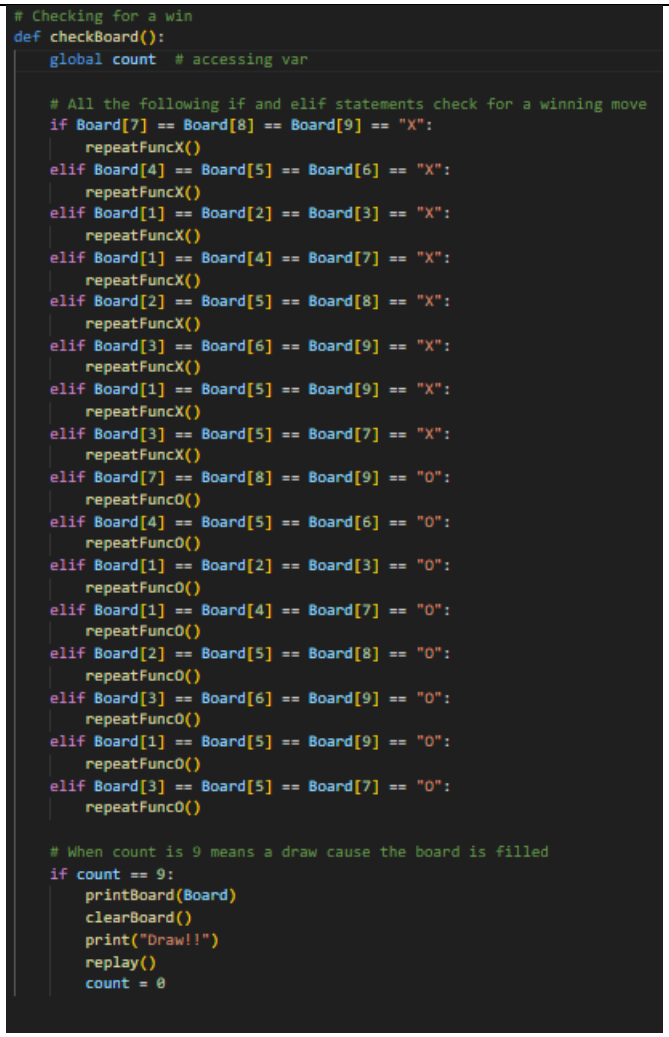
‘__init__(self, symbol)’: This is the constructor method of the class Token. It initializes a new instance of the Token class with the provided symbol. The symbol parameter represents the symbol associated with the token (e.g., "X" or "O")

The ‘move(self)’ method allows a token to make a move on the board. It also handles invalid values , updates the game board, and provides error handling for invalid moves.

```
class Token:
    def __init__(self, symbol):
        self.symbol = symbol

    def move(self):
        global count
        printBoard(Board)
        move = input(f"It is {self.symbol} turn. Move to which place (1-9): ")
        if move.isdigit(): # Method checks if input is valid
            move = int(move)
            if move in Board and Board[move] == " ":
                count += 1
                Board[move] = self.symbol
                return True
            elif move not in Board:
                print("Invalid move. Please pick a number between 1 and 9.")
                x_token.move()
            else:
                print("Place is already taken. Please pick an empty space.")
                x_token.move()
        else:
            print("Please enter a valid number.")
            x_token.move()
        return False
```

Instances of the ‘Token’ class are created for player X, Player O and the Computer. Each player is represented with one token, the reason ‘0_token’ and ‘Cpu_token’ are represented by the same symbol is because one will be used

```
x_token = Token("X")
o_token = Token("O")
cpu_token = Token("O")
```

| | |
|---|---|
| for Two-player whereas the other will be used for computer Player in Single player. | |
| This function keeps track of the current board and detects whether a player has made a winning move. If a player has won, it calls the function that announces the winner and ends the game.<br><br>The IF statement checks the current status of the Board, by referring to the key value pairs, and then checks if they are of the same Token. If this condition is met, either 'repeatFuncX' or 'repeeatFuncO' will be called.<br><br>There is also another IF statement that keeps track of the 'count' variable. This keeps count of moves made in the game. If 'count' is equal to '9', the game is ended in a draw.<br><br>Note: I now realised that I could have used a more efficient and practical approach for the IF statements. I could have used a for loop as well as a list that stores possible winning combinations. This could have saved time and improved code readability. |  <br>```<br># Checking for a win<br>def checkBoard():<br>    global count  # accessing var<br><br>    # All the following if and elif statements check for a winning move<br>    if Board[7] == Board[8] == Board[9] == "X":<br>        repeatFuncX()<br>    elif Board[4] == Board[5] == Board[6] == "X":<br>        repeatFuncX()<br>    elif Board[1] == Board[2] == Board[3] == "X":<br>        repeatFuncX()<br>    elif Board[1] == Board[4] == Board[7] == "X":<br>        repeatFuncX()<br>    elif Board[2] == Board[5] == Board[8] == "X":<br>        repeatFuncX()<br>    elif Board[3] == Board[6] == Board[9] == "X":<br>        repeatFuncX()<br>    elif Board[1] == Board[5] == Board[9] == "X":<br>        repeatFuncX()<br>    elif Board[3] == Board[5] == Board[7] == "X":<br>        repeatFuncX()<br>    elif Board[7] == Board[8] == Board[9] == "O":<br>        repeatFuncO()<br>    elif Board[4] == Board[5] == Board[6] == "O":<br>        repeatFuncO()<br>    elif Board[1] == Board[2] == Board[3] == "O":<br>        repeatFuncO()<br>    elif Board[1] == Board[4] == Board[7] == "O":<br>        repeatFuncO()<br>    elif Board[2] == Board[5] == Board[8] == "O":<br>        repeatFuncO()<br>    elif Board[3] == Board[6] == Board[9] == "O":<br>        repeatFuncO()<br>    elif Board[1] == Board[5] == Board[9] == "O":<br>        repeatFuncO()<br>    elif Board[3] == Board[5] == Board[7] == "O":<br>        repeatFuncO()<br><br>    # When count is 9 means a draw cause the board is filled<br>    if count == 9:<br>        printBoard(Board)<br>        clearBoard()<br>        print("Draw!!")<br>        replay()<br>        count = 0<br>```<br><br>These functions are used in the snippet above so I can call them instead of having to rewrite the same lines of code. When the dictionary 'Board' has three adjacent tokens, one of these functions are called by the 'CheckBoard' method.<br><br>The repeat functions print the status of the board and calls the winner, either X or O.<br><br>```<br># Simplifies code instead of repetition<br>def repeatFuncX():<br>    printBoard(Board)<br>    printWinnerX()<br><br><br># Simplifies code instead of repetition<br>def repeatFuncO():<br>    printBoard(Board)<br>    printWinnerO()<br><br><br># Printing win statements for X<br>def printWinnerX():<br>    print("Player X won!")<br>    print("End Game")<br>    replay()<br><br><br># Printing win statements for O<br>def printWinnerO():<br>    print("Player O won!")<br>    print("End Game")<br>    replay()<br>```<br><br>Note: At first, I had an issue where whenever Player X would win a game and they chose to restart, they were able to place 2 tokens in one go. As I was later developing the CPU's moves, I realised I had two Player X move functions being called one after another, one called from a restart function and the other at the start of the game. |

Once the replay function is called, it prompts the user for an input. If the user enters ‘y’ or ‘Y’, the ‘clearBoard’ function is called and Player X starts in the new game. However, if the user enters either ‘n’ or ‘N’, the users are transferred back to the main menu. If the user doesn’t enter an input other than these values, the replay function is recalled until they give a valid input.

```
# Replaying game
def replay():
    replayStr = str(input("Would you like to replay the game? (y/n): "))  # Getting input as a string
    if replayStr == "y" or replayStr == "Y":  # If input is Y or y
        clearBoard()
        x_token.move()
    elif replayStr == "n" or replayStr == "N":  # If input is N or n
        main_menu()
        quit()  # Ends programme
    else:
        # If input is not Y or N
        print("Invalid input, Try again!")
        replay()
```

This function uses a while loop to iterate through all the values on the ‘Board’ dictionary and replaces them with empty spaces, essentially restarting the game board. It stop iterating after the 9th time.

```
# Clearing the board
def clearBoard():
    global count  # Accessing count
    count = 0  # Set count to 0 to restart

    # while loop to simplify code
    i = 1
    while i < 10:  # While i is less than 10
        Board[i] = " "  # Set the value of i of board to " " which means emtpy
        i += 1
    # Repeats 9 times
```

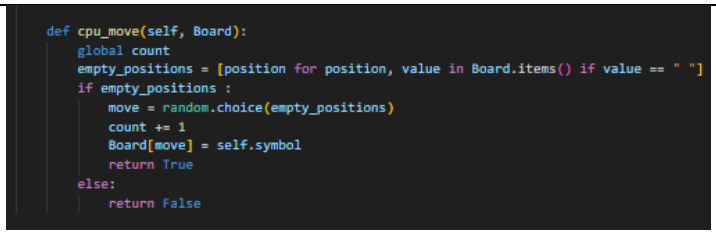
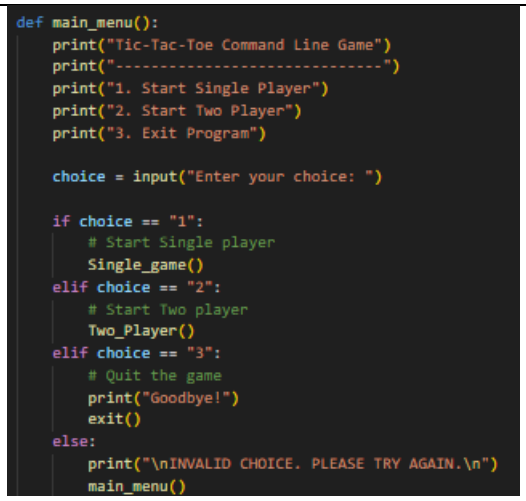
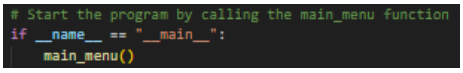
The Two_Player() function represents a game mode for two players. It starts by displaying a message and showing the coordinates of the game board. Inside a continuous loop, the players take turns making moves. After each move, the board is checked for a win condition using the checkBoard() function. If all moves have been made and no winner is found, it declares a draw and asks if the players want to replay. The loop continues until the game is over, and the count variable is reset to 0 for the next game.

```
def Two_Player():
    # Continuous loop
    print("\nYou chose Two player\n")
    display_coordinates()
    global count
    while True:
        # For start
        if x_token.move():
            checkBoard()
        o_token.move()
        checkBoard()
        if count == 0:
            x_token.move()

        # Checking for a draw
        if count == 9:
            clearBoard()
            print("Draw!!")
            replay()
            count = 0
```

Note: I chose to start with the Two player function of the program before Single Player. This was because I thought it would be easier to code the functions for two real players first so I can then reuse the same code frame with Single player but instead implement the random choice generator for the computer.

The Single_game() function represents a single-player game mode. It prompts the player's move and checks for a win condition. Then, the computer makes its move based on the ‘cpu_move’ function and checks for a win. If no moves have been made yet, the player is prompted again. After each move, it checks for a draw condition. If no winner is found, it clears the board and asks if the players want to replay. The game continues until completion,

```
def Single_game():
    print("\nYou chose Single player\n")
    display_coordinates()
    global count
    while True:
        # For start
        if x_token.move():
            checkBoard()
        print("It is now Computer's turn")
        cpu_token.cpu_move(Board)
        checkBoard()
        if count == 0:
            x_token.move()

        # Checking for a draw
        if count == 9:
            clearBoard()
            print("Draw!!")
            replay()
            count = 0
```

| | |
|---|---|
| with the count variable reset for the next game. | |
| The 'CPU_move' method is within the Token class. It takes 'self'(symbol) and the Board dictionary as arguments. The Board is checked for any empty key value pairs and stores it in the 'empty_positions' list. A randint function is generated to pick a coordinate for the Computer. The overall move count is incremented by 1 and the computer's token coordinate is reassigned to the current game board. |  |
| The main_menu() function displays the title and menu options for a Tic-Tac-Toe command line game. It prompts the user to choose between single-player, two-player, or exiting the program. Based on the user's input, the corresponding game mode is executed, or the program is exited. If an invalid choice is entered, an error message is displayed, and the menu is shown again. The function keeps looping until a valid choice is made, ensuring smooth navigation through the game options. |  |
| When the main script is running. It will call the "main_menu" function. This is what starts the program |  |

## Implementation reflection

Coding a Tic-Tac-Toe command line game in Python was an engaging and educational experience. Building the game required careful consideration of the game's rules, user interactions, and logical checks for win conditions and draws. It was important to design the code in a modular and organized manner to ensure readability and maintainability.

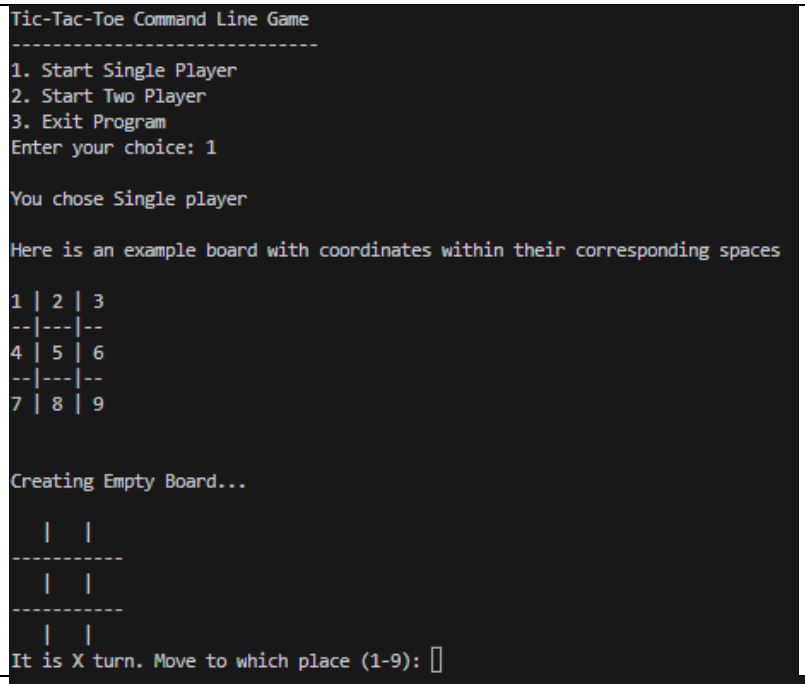
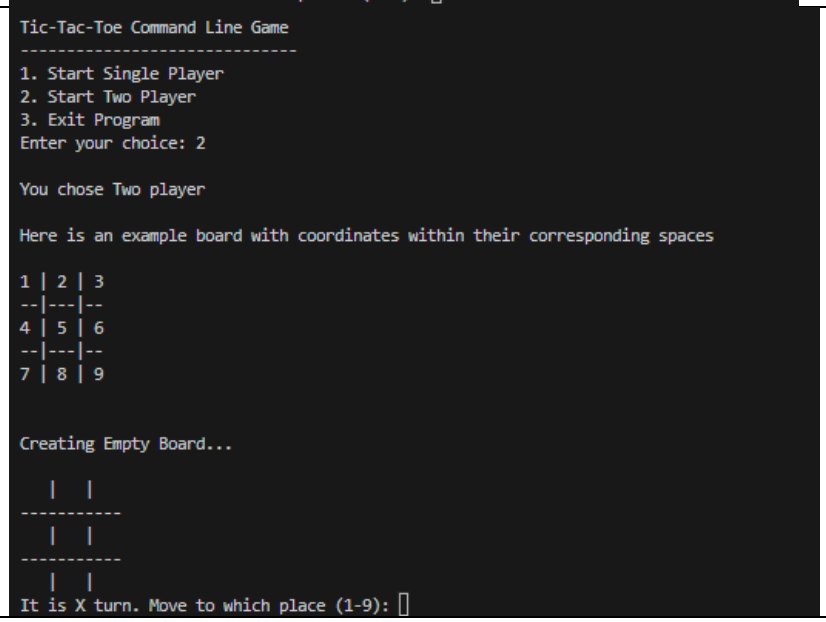
When I started the coding of this program, I initially used letters as coordinates instead of numbers they were their own classes. After hours of trying to figure out how to use them, I gave up and instead used numbers to represent the empty spaces within a dictionary.

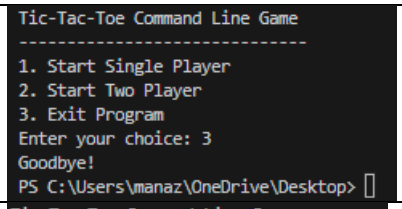
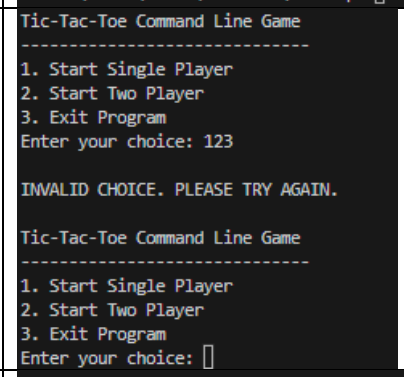
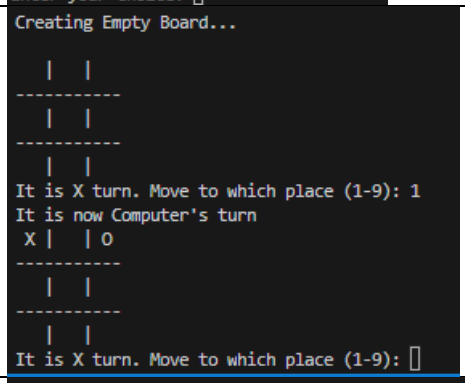
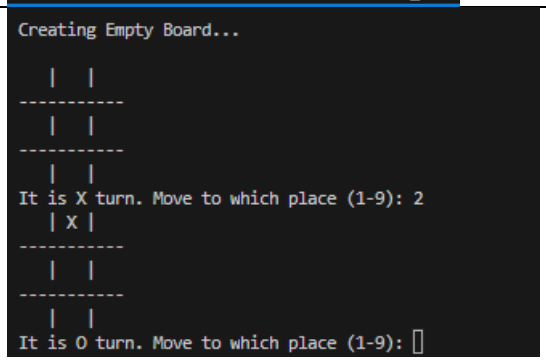
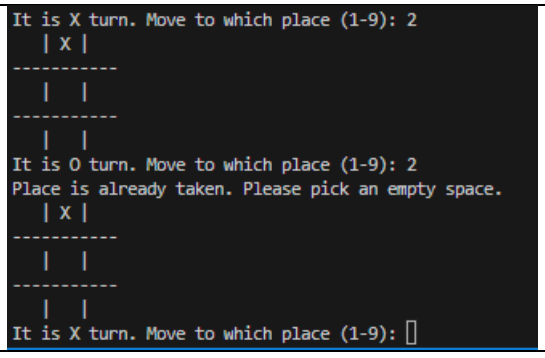
Overall, creating this program deepened my knowledge on control flow structures, input validation, and utilizing function and custom types to structure the code. I believe there are still areas I need to improve on such as the reusability of code as there are parts of the program where I used the same lines of code over and over when I could have used loops. This project has enhanced my problem-solving skills and reinforced the importance of planning and organising code.

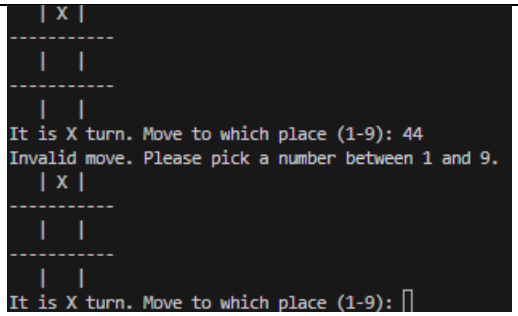
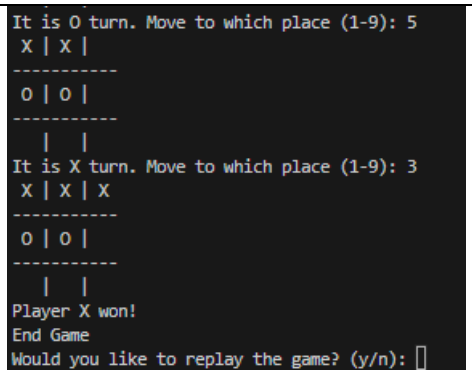
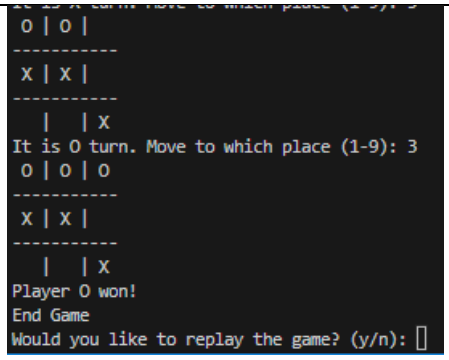
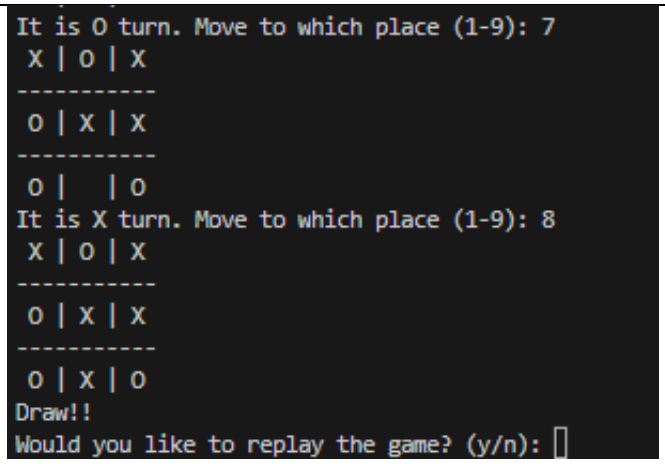
# Testing

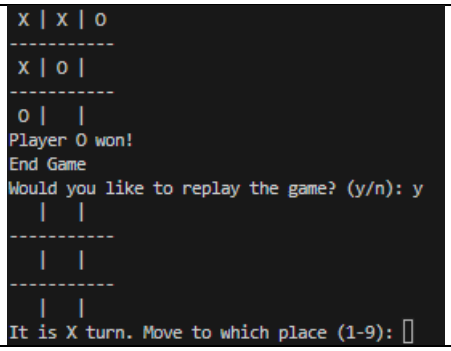
**Manual testing:**

Manual testing refers to the process of manually executing test cases to identify bugs or issues in a program. While it can be useful for small-scale testing, manual testing becomes impractical and time-consuming when dealing with complex or large software systems, as it is prone to human errors, lacks repeatability, and hinders efficient test coverage (Guru99.com, 2019).

| Test Case ID | Test Case Description | Input Data | Expected Result | Screenshots | Pass/Fail |
|---|---|---|---|---|---|
| Test 1 | Starting a single player game from the main menu | Enter '1' key | Single player game will start up with a new single player game board and example game board. Player X starts |  | Pass |
| Test 2 | Starting a two-player game from the main menu | Enter '2' key | Single player game will start up with a new Two player game board and example game board. Player X starts |  | Pass |

| | | | | | |
|---|---|---|---|---|---|
| Test 3 | Exiting the program from the main menu | Enter ‘3’ | Program ends |  | Pass |
| Test 4 | Entering a value which isn’t one of the options | Enter ‘123’ | It prints an invalid input statement, and the user is asked to try again |  | Pass |
| Test 5 | Computer Placing a token on the game board | Random generated number | The computer will place a token on the random generated coordinate |  | Pass |
| Test 6 | Real life Player Placing a token on the game board | Enter ‘2’ | The player’s token should be placed on the coordinate 2 of the game board |  | Pass |
| Test 7 | Placing a token in an occupied space on the grid | Enter ‘2’ again | There will be an invalid input statement and the player will be asked to retry |  | Pass |

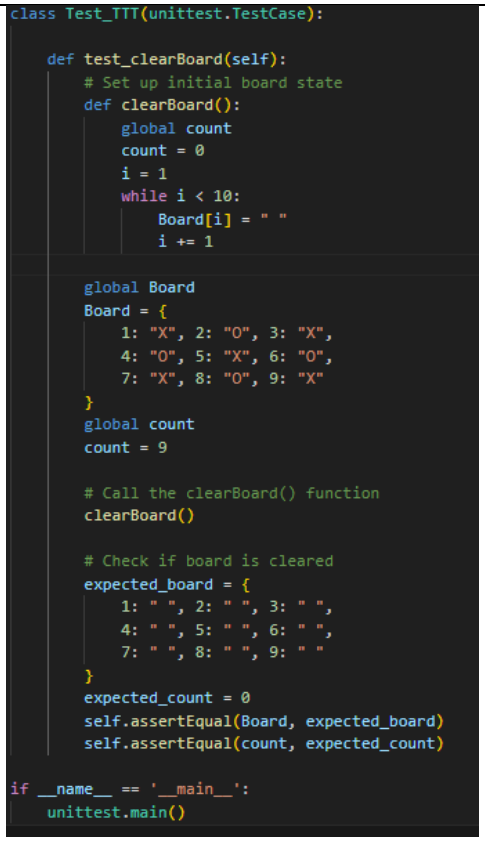
| | | | | | |
|---|---|---|---|---|---|
| Test 8 | Specifying an invalid coordinate | Enter ‘44’ | There will be an invalid input statement and the player will be asked to retry |  | Pass |
| Test 9 | Player X places a game winning move | Enter game winning coordinate | The Game comes to an end by stating Player X as the winner. There will be an option to replay |  | Pass |
| Test 10 | Player 0 places a game winning move | Enter game winning coordinate | The Game comes to an end by stating Player O as the winner. There will be an option to replay |  | Pass |
| Test 11 | Game ends with a draw | Entering a move that makes ‘count == 9’ | There will be a game board displayed with no available spaces. The game will come to and end where the program states the game has come to a draw. There will be an option to replay |  | Pass |

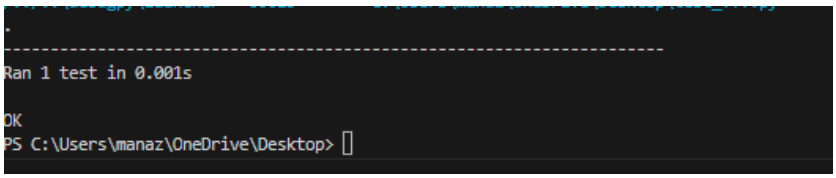
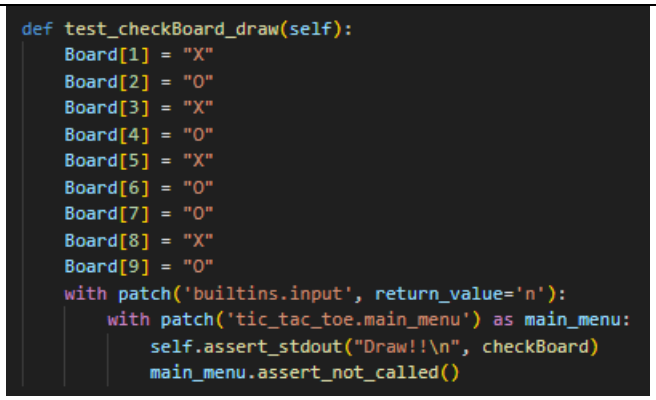
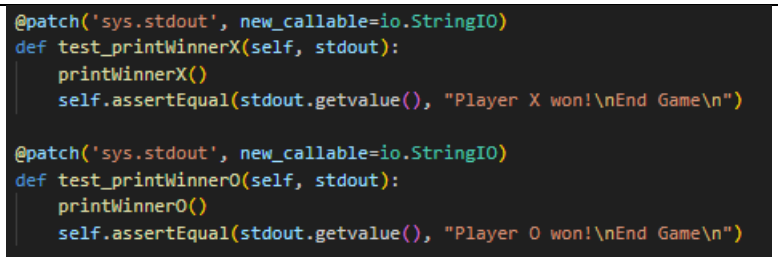
| | | | | | | |
|---|---|---|---|---|---|---|
| Test 12 | Restart Game | Enter ‘y’ or ‘Y’ | The game should restart, displaying a new clear game board |  | | Pass |

**Automated testing:**

Automated testing involves using software tools to execute test cases and compare actual results with expected outcomes. It offers practicality by enabling faster test execution, repeatability, wider test coverage, and the ability to detect issues early in the software development lifecycle, leading to improved efficiency and product quality. One limitation of automated testing is that it may not effectively detect logical errors or flaws in the systems build.

## Unit Testing

| Test Description | Test code |
|---|---|
| This test checks if the clearBoard() function returns an empty board as well as resetting the count variable to 0. The Values of the Board as well as the count variable are run through the test data and the actual output is compared to the expected output.<br><br>I kept coming across problems with the test failing and after trying different solutions, I concluded that there was something wrong with how I imported the TTT command line file. To fix this issue, I just copied and pasted the function from the TTT |  |

| | |
|---|---|
| program file too the test case file and the test worked. This could indicate that another part of my code could be interfering with the state of the game board which isn't giving the required output. |  |
| The provided test case examines the behaviour of the checkBoard function within a tic-tac-toe game. The scenario being simulated is one where the game board is filled with X and O moves, resulting in a draw. By patching the input function to return 'n' and using assert_stdout, the test ensures that when checkBoard is executed, it correctly prints "Draw!!" to the standard output. Additionally, the test case uses assert_not_called to verify that the main_menu function is not invoked, indicating that when a draw occurs, the checkBoard function should not navigate back to the main menu. |  |
| These test cases validate the behaviour of the printWinnerX() and printWinnerO() functions by confirming that they produce the expected output when called. By capturing the printed output and comparing it to the expected value, the tests ensure that the functions correctly print the winner message and end the game accordingly. |  |

| | |
|---|---|
| The assert_stdout method is a custom method that simplifies testing functions that generate output to the standard output. It captures the output, compares it to the expected output, and performs the necessary assertion. By using this method, it becomes easier to verify the behaviour of functions that produce textual output | `@patch('sys.stdout', new_callable=io.StringIO)`<br>`def assert_stdout(self, expected_output, func, *args):`<br>`  func(*args)`<br>`  self.assertEqual(expected_output, self.stdout.getvalue())` |
| These test cases verify that the checkBoard function correctly detects winning conditions for both Player X and Player O. They validate the printing of the appropriate winner message and confirm that the game does not navigate back to the main menu after a win. | `def test_checkBoard_X_wins(self):`<br>`  Board[1] = Board[2] = Board[3] = "X"`<br>`  with patch('builtins.input', return_value='n'):`<br>`  with patch('tic_tac_toe.main_menu') as main_menu:`<br>`  self.assert_stdout("Player X won!\nEnd Game\n", checkBoard)`<br>`  main_menu.assert_not_called()`<br><br>`def test_checkBoard_O_wins(self):`<br>`  Board[4] = Board[5] = Board[6] = "O"`<br>`  with patch('builtins.input', return_value='n'):`<br>`  with patch('tic_tac_toe.main_menu') as main_menu:`<br>`  self.assert_stdout("Player O won!\nEnd Game\n", checkBoard)`<br>`  main_menu.assert_not_called()` |

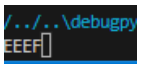
Running these unit tests, after hours of trying to get them to pass, I gave up as I kept on encountering the same errors. When reading through the errors, it was telling me that the Game board wasn't being cleared even though it was. At first, I thought I didn't import the program file properly however after long hours of trying to get it to work, it seemed like the tests weren't passing because there may have been code from another function of the program that was accessing and changing it.

Also, when I copied and pasted the function I wanted to test into the test file, the test had passed. This could mean that I had either imported the file incorrectly or there is another part of my program that's changing the state of the game board when it isn't supposed to.

Here is the traceback I got:

```
AssertionError: {1: 'X', 2: 'O', 3: 'X', 4: 'O', 5: 'X', 6: 'O', 7: 'X', 8: 'O', 9: 'X'} != {1: ' ', 2: ' ', 3: ' ', 4: ' ', 5: ' ', 6: ' ', 7: ' ', 8: ' ', 9: ' '}
- {1: 'X', 2: 'O', 3: 'X', 4: 'O', 5: 'X', 6: 'O', 7: 'X', 8: 'O', 9: 'X'}
?     ^     ^     ^     ^     ^     ^     ^     ^     ^

+ {1: ' ', 2: ' ', 3: ' ', 4: ' ', 5: ' ', 6: ' ', 7: ' ', 8: ' ', 9: ' '}
?     ^     ^     ^     ^     ^     ^     ^     ^     ^
```

Out of the 6 tests, 1 came up as failure, 3 had errors, and the other 2 weren't showing up. A reason for them not being able to show up could be because there is an issue with the test runner or a problem with importing the TicTacToe file. Here is a snippet of the output:




## Reflective Summary

The process of creating and testing the TicTacToe program involved manual testing for initial validation and unit testing for thorough verification. While manual testing initially indicated the program's correctness, unit testing exposed errors and discrepancies. This highlighted the importance of unit testing as a critical step in software development, allowing for systematic and automated validation of program behaviour. The failures encountered during unit testing provided valuable insights into the program's weaknesses, enabling targeted debugging and refinement to improve its overall quality and reliability.

After a thorough review of my code and the test cases, I gained an idea on why I went wrong with my unit testing, I have now learnt that I need to code in a more efficient way to ensure the robustness of the program. To do this I need to thoroughly go through my code to make sure each function is doing what it needs to and isn't affecting anything else in the program. This will be done in my spare time after the due date of this assignment as I am running out of time.

## Version Control:

Python Koans aim is to enhance programmers' understanding of Python through practical exercises and test-driven learning. Providing documentation to contributors in a software project offers several benefits. Firstly, it reduces the learning curve by providing clear instructions and guidelines, facilitating on boarding, and grasping project procedures. Secondly, it enhances code comprehension, promoting consistency and integrity. Additionally, it promotes collaboration, establishes coding standards, and supports maintenance and debugging efforts. Moreover, it serves as a knowledge repository, ensuring project continuity and empowering new team members. In summary, documentation improves onboarding, comprehension, collaboration, maintenance, and project knowledge (Atlassian, 2019).

## Clone:

If someone wants to contribute to a project, you clone the files to their local computer. This creates a local copy of the program which you can edit. When you clone a project, you download the entire repository's history, including branches, commits and files to your local machine. Cloning is an essential step when you want to contribute to an open-source project and collaborate with others. Once you have either made changes, created new branches, and commit modifications you can push your changes back to the remote repository if you have necessary permissions (blog.hubspot.com, n.d.).

## Commit:

Commits capture and record changes to a project. It enables tracking, reverting, collaboration, code review and merging workflows. If a bug arises after a specific commit, then you can revert to those changes, making it quick and easy. When changes have been made and you commit, it's only saved locally until you eventually push it to the remote repository (GitHub, n.d.).

## Pull:

The "Pull" command is used when one or more contributors have made changes to the remote repository. It fetches the latest changes from the remote repository and merges them into the local repository. It updates the local repository with the latest commits made by other contributors. Pulling ensures that a developer's local copy is up to date with the changes made by fellow developers before they continue to work on the project (Atlassian, n.d.).

## Push:

"Pushing" refers to sending local commits to a remote repository. For example, if you have cloned a remote repository and made changes you can then push those changes back to the remote repository with the necessary permissions. When a developer pushes their changes, they make them available to others working on the project. This promotes collaboration and allows for people to fix bugs and other errors (GitHub, n.d.).

## Merge:

The merge command is used to integrate changes together into a single branch. By combining different lines of development, it brings together changes made in different branches and creates a new commit that incorporates those modifications. It is essential for integrating individual contributions into a shared codebase as well as ensuring that changes from multiple contributors coexist harmoniously. For example, if a developer has added a new feature to a branch, it can be integrated via merging (The Mergify Blog, 2021).

## Branch:

A branch is a separate version of the main repository. It allows developers to work on features or fixes without impacting the main codebase. This enables parallel development and isolates changes until they are ready to be integrated. Each branch has its own commit history which allows developers to work on different features simultaneously and merge them back to the main repository when completed (GitHub Docs, n.d.).

## Pull Request:

A pull request is a request made by a developer to merge their changes from a branch into another branch, often the main branch. It provides a platform for discussions, feedback, and validation before the changes are merged, promoting code quality and collaboration within a team (GitHub Docs, n.d.).

## Benefits of version control systems to developers

**Collaboration** - version control enable multiple developers to work on the same codebase simultaneously, allowing changes to be tracked and resolving conflicts. Merging files after changes have been made are easier and cleaner to handle as everyone has their own local copies to which they can add to the main repository (Atlassian, 2019).

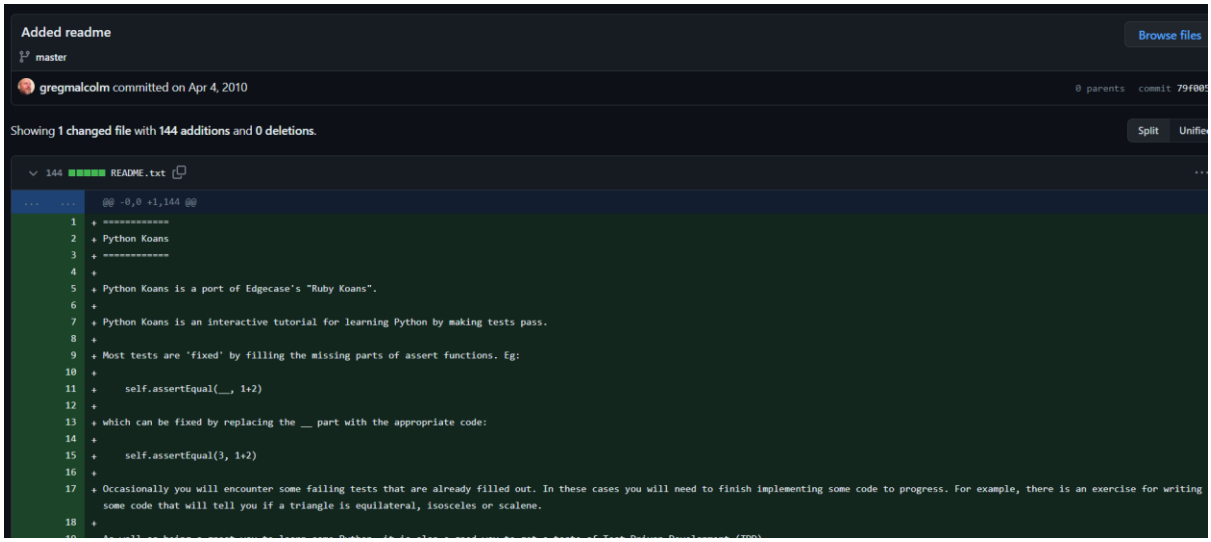
**Version history** – The ability to access and roll back to earlier versions of the code gives developers a safety net and makes it easier to troubleshoot and fix bugs. If there were issues with the current version, you can load the previous version to remove and re-implement the feature once it's working correctly (Atlassian, 2019).

**Branching and Parallel Development** - Branching is a feature of version control systems that enables developers to work on distinct features or experiments without affecting the main codebase. This encourages parallel development and makes feature isolation easier (Atlassian, 2019).

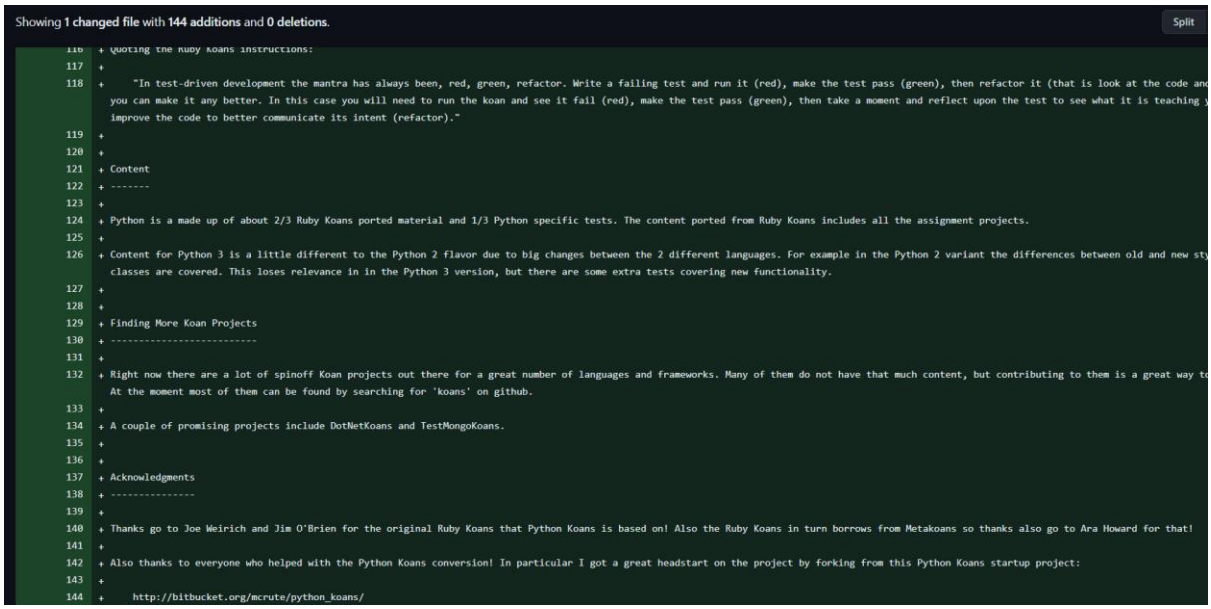
**Code Review** - Version control systems support code review workflows, allowing team members to review, comment, and provide feedback on potential changes before merging them (Atlassian, 2019).

# Python Koans' Commit History

## First Major Commit:

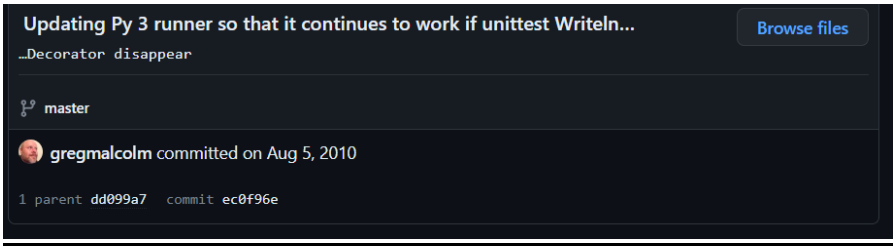


As you can see from the snippet above, one of the first major commits was made during the 4th April 2010 by Greg Malcolm, the creator of Python Koans. This was with the change of 1 file , the README file, where 144 additions were made with no deletions. You can see all the information about the commit such as the ID, the date it was made, what was added, what was deleted, and the Author.




There were 144 lines added to the README file. The significance of this file is to provide details on how to get started with the Python Koans project and any additional information provided by the developers.

## Commit with relevant change:




As you can see from the snippet above, a commit with a relevant change was made on the 5th August 2010 by Greg Malcolm again. This change was made to update the “Python 3” files so that the program would still work without the use of the “unittest” module.

```
 7   7    from . import path_to_enlightenment
 8   8    from .sensei import Sensei
     9  + from .writeln_decorator import WritelnDecorator
 9  10
10  11    class Mountain:
11  12        def __init__(self):
12      -         self.stream = unittest._WritelnDecorator(sys.stdout)
        13 +      self.stream = WritelnDecorator(sys.stdout)
13  14            self.tests = path_to_enlightenment.koans()
14  15            self.lesson = Sensei(self.stream)
15  16
```

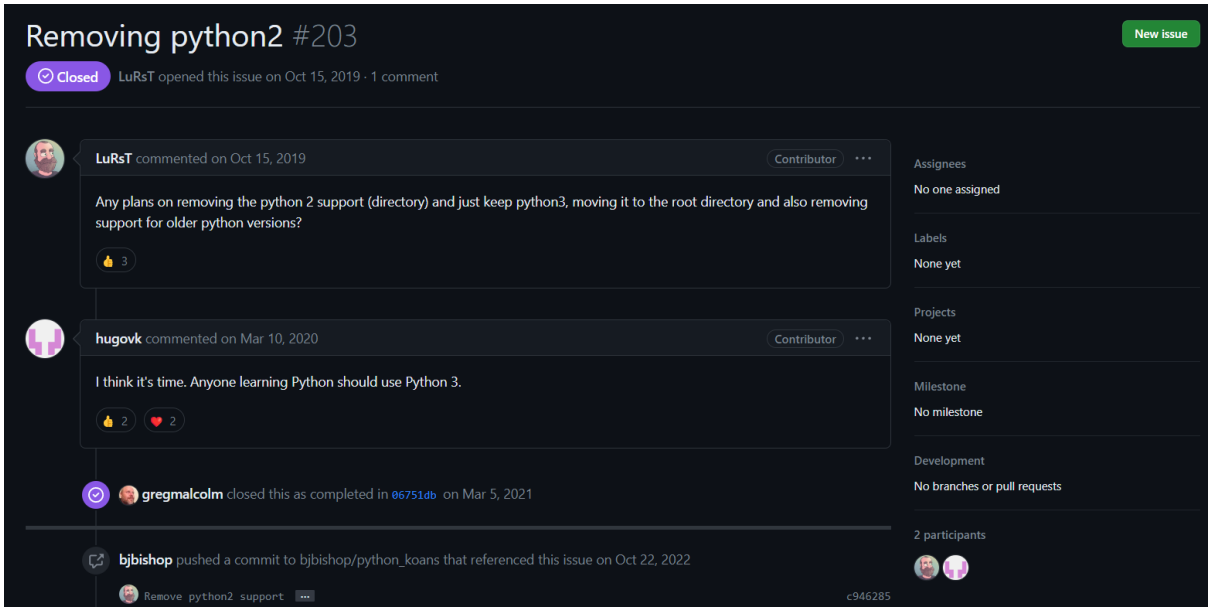
As you can see from snippet above, the line of code assigning the 'self.stream' to the instance of the 'WriteLnDecorator' class from the Unit test was removed and instead a new line of code was added which assigned an instance of another module to it.

## Second File Update

```
3 python 3/runner/runner_tests/test_sensei.py
...
    @@ -8,6 +8,7 @@
 8   8  from libs.mock import *
 9   9
10  10  from runner.sensei import Sensei
    11 + from runner.writeln_decorator import WritelnDecorator
11  12  from runner.mockable_test_result import MockableTestResult
12  13
13  14  class AboutParrots:
    @@ -83,7 +84,7 @@ class AboutFreemasons:
83  84  class TestSensei(unittest.TestCase):
84  85
85  86      def setUp(self):
86      -         self.sensei = Sensei(unittest._WritelnDecorator(sys.stdout))
    87 +         self.sensei = Sensei(WritelnDecorator(sys.stdout))
87  88          self.sensei.stream.writeln = Mock()
88  89
89  90      def test_that_it_delegates_testing_to_test_cases(self):
```

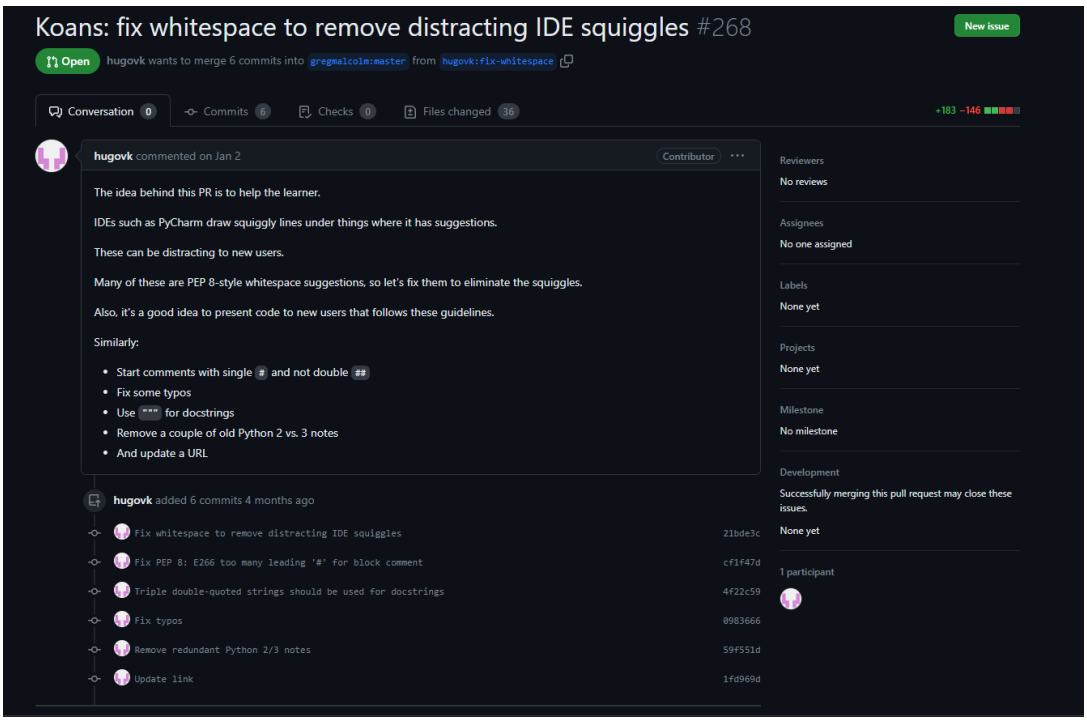
The 'test_sensei.py' file was edited where there were two additions and 1 deletion. These were the same changes made to the first file where the unit test module was removed. This commit was made within the main branch or the repository.

## Issue Tracker:




The snippet above shows a closed request made on the issue tracker by a user named LuRsT on the 15th October 2019. This was a request made to remove the previous python directory as there was a newer, more updated one which python learners could use. This request was closed by Greg Malcolm, the main developer, as he later on removed the Python 2 directory along with the older python versions. This issue was necessary as learners that hadn't known there was a new version would have still been using the old one which may have had bugs.

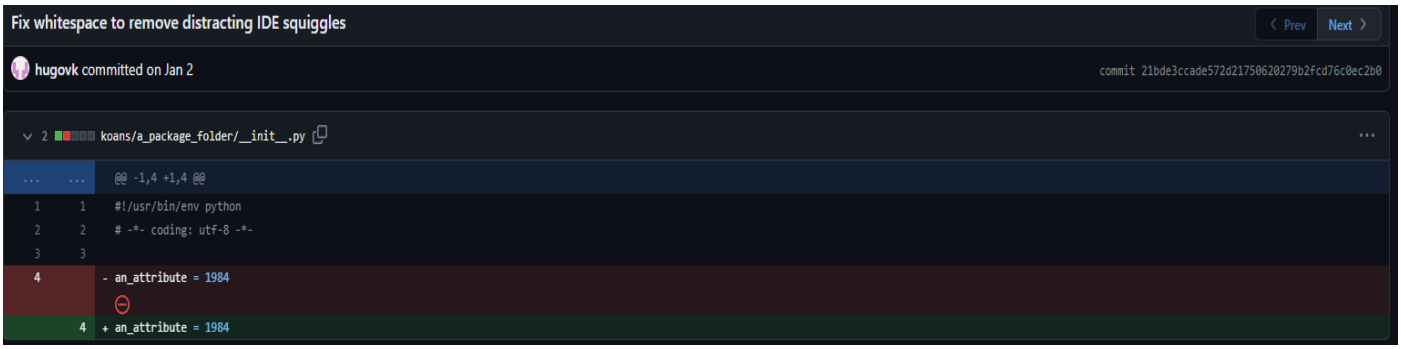
## Pull Request Tracker:




The Pull Request above was made by Hugovk on the 2nd January 2023 where there were 6 commits made that were merged. The users' intention was to make the UI cleaner and simpler for new learners that were using the software. 34 files were changed where edits such as lines with "##"

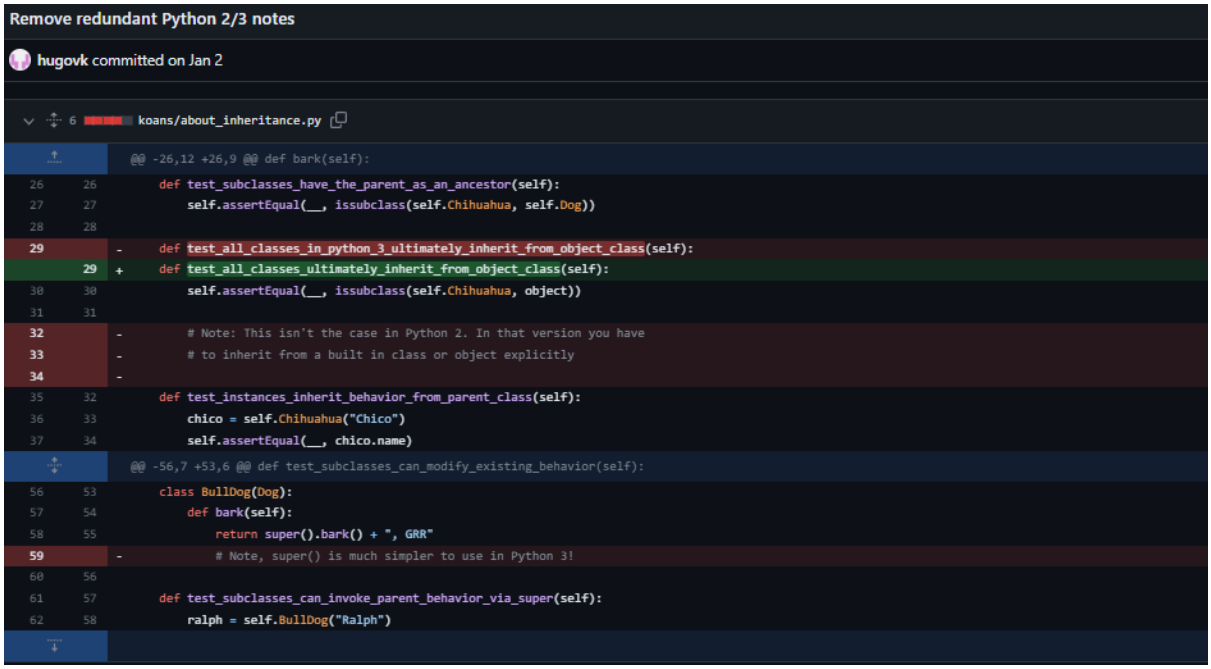
were changed to “#”, fixing typos, and removing whitespace. On the bottom half of the snippet you can see all the commits made with their commit ID’s to the right of them.

Here are a few files that were edited:




You can see a simple change was made where there was one line removed with white space and replaced with another line with the same text. You can also see the commit Id on the top right as well as who made the commit and what date it was made.


Here was another file that where there were 44 additions and 44 deletions. The lines with double comments were removed and instead replaced with single comments. This change was made to prevent confusion for learners.




Here is a Koan that was edited where unnecessary comments were removed as well as the name of a testing function. All these changes were made to 34 files altogether to make it so every koan was up to date and they weren't too visually overwhelming for learners.

## How Could I have used version control such as Git to aid me with my previous task for this project?

I could use Git in many ways to aid me throughout the development of this project. If I had a repository, I could have other developers read through my code and make commits to the repository which could improve the readability as well as increasing the robustness of my code. The errors that I had made could be read and fixed by someone else. They would discuss where I went wrong and how they fixed it, creating a healthy learning environment for everyone working on the project. I think I could have also received a lot of help and fixes on the unit testing as I did struggle with identifying why they didn't pass.

With each developer being able to have their own local repository, they could develop their own way of coding the functions. Developers can then come together to discuss their code and see who has the most efficient and robust code, and see how they could improve their own way of coding.

# References:


- Atlassian (2019). What is version control. [online] Atlassian. Available at: https://www.atlassian.com/git/tutorials/what-is-version-control
- blog.hubspot.com. (n.d.). How to Clone a GitHub Repository: A Step-By-Step Guide. [online] Available at: https://blog.hubspot.com/website/clone-github-repository#why
- GitHub. (n.d.). Git Guides - git commit. [online] Available at: https://github.com/git-guides/git-commit
- Atlassian (n.d.). Git Pull | Atlassian Git Tutorial. [online] Atlassian. Available at: https://www.atlassian.com/git/tutorials/syncing/git-pull#:~:text=The%20git%20pull%20command%20is
- GitHub. (n.d.). Git Guides - git push. [online] Available at: https://github.com/git-guides/git-push
- The Mergify Blog. (2021). Everything You Need to Know About Git Merge. [online] Available at: https://blog.mergify.com/everything-you-need-to-know-about-git-merge/
- GitHub Docs. (n.d.). About branches. [online] Available at: https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-branches
- GitHub Docs. (n.d.). About pull requests. [online] Available at: https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests
- Paredes, J.H.G. (2021). Specifications for Developers: Starting with Gherkin-Part 1. [online] Medium. Available at: https://dezkareid.medium.com/specifications-for-developers-starting-with-gherkin-part-1-67c2c3b63dd7#:~:text=To%20write%20specifications%20we%20need
- Guru99.com. (2019). Automation Testing Vs. Manual Testing: What's the Difference? [online] Available at: https://www.guru99.com/difference-automated-vs-manual-testing.html