**Initial Setup**

The project will be developed on a Windows desktop system, with Visual Studio Code serving as the Integrated Development Environment (IDE). Unlike working with C++, this project will be implemented using Python. Python offers a simpler and more flexible syntax compared to C++, making it easier to write and maintain code. Visual Studio Code provides a powerful and user-friendly environment for Python development, offering features such as code navigation, debugging, and integrated Git support. With Python's extensive libraries and Visual Studio Code's robust features, the development process is expected to be efficient and streamlined. Additionally, Python's platform independence ensures that the project can be easily deployed across different operating systems.

**Project Overview**   
  
The Connect4 game is a classic board game where players aim to form a line of four tokens horizontally, vertically, or diagonally. This Python implementation allows two players to play against each other or against the computer. Players take turns dropping tokens into a grid, aiming to outmaneuver their opponent and achieve victory. The game is played on a 6x7 grid, and the first player to connect four tokens in a row wins.

For the Connect4 project, the requirements analysis outlines the key features and functionalities of the game:

**Gameplay Mechanics:**

In the game, players take turns placing their tokens into a grid with 6 rows and 7 columns. The objective is to create a line of four tokens either horizontally, vertically, or diagonally before the opponent does.

Players can compete against each other in a two-player mode or play against the computer.

To make a move, players specify the row and column where they want to place their token. For example, a player might input "Row 3, Column 5" to place their token in the third row and fifth column of the grid. The game then updates the grid accordingly, and play proceeds to the next player's turn.

The game continues until one player successfully connects four tokens in a row, or until the grid is full without a winner, resulting in a draw. This game offers a strategic challenge and can be played either against friends or against the computer for a single-player experience.

**Game Flow:**

The game starts with a welcome message and prompts players to enter their names and choose their tokens (X or O).

Players alternate turns until one player wins or the game ends in a draw.

**Input Validation:**

Input validation ensures that players enter valid moves within the boundaries of the game board and into empty cells.

**Winning Condition:**

The game checks for the winning condition after each move to determine if a player has won.

**End Game Conditions:**

The game ends when one player achieves four tokens in a row or when the game board is full (resulting in a draw).

**Player Interaction:**

Players receive feedback after each move, informing them of the success or failure of their move.

**CPU Player (Optional):**

The game includes an option for a single-player mode where the player competes against the computer.

The CPU generates random moves as its strategy.

**Visual Representation:**

The game displays the current state of the board after each move, allowing players to visualize the game progress.

ASCII art can be used to enhance the visual representation of the game board.

**Behavior Driven Development (Gherkin Specifications)**

|  |
| --- |
| Feature: Playing Connect4 Game |
| As a CLI USER/PLAYER, I want to enter playe game |
| Scenario: Player starts the game |
| Given the game has started When the game initializes  Then the game displays the current board |
| Scenario: Player makes a valid move |
| Given it is player's turn  When the player selects a valid column to drop the token  Then the token is placed in the chosen column |
| Scenario: Player makes an invalid move (column full) |
| Given it is player's turn  When the player selects a column that is already full  Then the game displays a message indicating the move is invalid |
| Scenario: Player wins horizontally |
| Given the player has placed four tokens in a row horizontally  Then the game declares the player as the winner |
| Scenario: Player wins vertically |
| Given the player has placed four tokens in a column vertically  Then the game declares the player as the winner |
| Scenario: Player wins diagonally (positive slope) |
| Given the player has placed four tokens diagonally in a positive slope  Then the game declares the player as the winner |
| Scenario: Game ends in a draw |
| Given the game board is full and no player has won  Then the game declares a draw |
| Scenario: CPU makes a valid move |
| Given it is CPU's turn  When the CPU selects a valid column to drop the token  Then the token is placed in the chosen column |
| Scenario: CPU makes an invalid move (column full) |
| Given it is CPU's turn When the CPU selects a column that is already full  Then the game displays a message indicating the move is invalid |
| Scenario: CPU wins horizontally |
| Given the CPU has placed four tokens in a row horizontally  Then the game declares the CPU as the winner |
| Scenario: CPU wins vertically  Given the CPU has placed four tokens in a column vertically  Then the game declares the CPU as the winner |
| Scenario: CPU wins diagonally (positive slope) |
| Given the CPU has placed four tokens diagonally in a positive slope  Then the game declares the CPU as the winner |
| Scenario: CPU wins diagonally (negative slope) |
| Given the CPU has placed four tokens diagonally in a negative slope  Then the game declares the CPU as the winner |

Data Model

**Input:**

* User Input (Standard Input)

**Output Messages:**

* Welcome to the Connect 4 game!
* Current Player: [Player Name]
* [Connect4 Board Display]
* Invalid move! Please choose an empty cell.
* Invalid input! Please enter numbers.
* [Player Name]'s token is [Player Token]
* [Player Name]'s turn ([Player Token]):
* [Player Name] wins!
* It's a draw!

**Error Handling:**

* Invalid input! Please enter a number.
* Invalid token! Please enter X or O.

Name Model: In the Connect 4 game, the name model represents the names chosen by the players. Each player selects a unique name as their identifier for the game. The name type contains the set of names chosen by the players, ensuring that each player is identified uniquely.

Let Name be the set of player names:

Name={Name1 , Name2​}

Here, each Name represents a player's chosen name, making the name effectively a subset of all possible player names. This approach simplifies the validation of inputs, ensuring that only valid player names are used during gameplay.

For example:

Name={"Player1","Player2"}

In this case, the 𝑁𝑎𝑚𝑒Name set has a cardinality of 𝑛n, representing the number of players in the game. Each element of the set is a unique player name, ensuring that the name remains valid and constrained to the selected player names.

Turn Model: In the Connect 4 game, turns are organized as a sequence of players, where each player takes their turn in order. The active player, who is currently taking their turn, is the player at the head of the sequence.

The player sequence can be represented as follows:

Let Players be a sequence of players:

Players = seq < Player1 , Player2>

Let PPP represent the set of all players. In this case,

P={Player1,Player2}

P = { {Player1}, {Player2} }

P={Player1,Player2}.

The sequence Players {Players}Players can be defined as an ordered pair from this set.

Here’s a more formal way to express it:

Players ∈ P^2

where P^2 is the Cartesian product of PPP with itself, representing all possible ordered pairs of players from the set PPP.

**Axiomatic Definitions and Functions**

Axiomatic Definitions:

**Player Model:** In the Player Model for Connect 4, each player serves as one of the two opponents engaged in the game. The Player Model encompasses various attributes and functionalities tailored to manage player interactions within the game environment.

The primary attribute of the Player Model is the "Name," which serves as the unique identifier for each player. Players can choose their own names, providing a personalized touch to their gaming experience. The name not only distinguishes one player from another but also helps in tracking their progress and actions throughout the game.

Another key aspect of the Player Model is the concept of the "Inventory," although not directly applicable in Connect 4. Unlike games with inventories, such as role-playing games, where players collect and manage items, Connect 4 doesn't involve the use of items. Instead, the focus is entirely on strategic gameplay, where players aim to connect their tokens to win the game. Therefore, the Inventory attribute remains unused in this context.

Moving on to the Game Model, it serves as the backbone of the Connect 4 game, orchestrating the flow of the game and facilitating interactions between players and the game environment.

The "Current Room" attribute, a common feature in many game models, is not applicable in Connect 4 since the game doesn't have distinct rooms or areas to navigate. Instead, players interact directly with the game board, which consists of a grid of cells where tokens are placed.

Similarly, the "Collection of Rooms" attribute, often found in games with multiple locations or levels, is not relevant in Connect 4. The game is played on a single game board, eliminating the need for separate rooms or environments. Therefore, the focus of the Game Model in Connect 4 is on managing the game board, tracking player moves, and determining the outcome of each game session.

**Functions**:

**start\_game(Game):** Initializes the Connect 4 game by setting up the board and starting the gameplay loop.

**move(Game):** Handles player moves in Connect 4 by adding tokens to the board.

**play(Game):** Main loop of the Connect 4 game where players take turns placing tokens on the board until there is a winner or a draw. Handles player input, moves, and game state changes.

T2 Implementation

Now that we're ready to implement the program, we'll distinguish between two types of functions: pure and impure. Pure functions only operate within their scope and don't modify any external program state, while impure functions do. Additionally, functions can be categorized as total or non-total. Total functions cover all possible input-value pairs, while non-total functions don't. We'll aim to create pure, total functions wherever possible.

**Connect4 Class:**

class Connect4:

    def \_\_init\_\_(self):

        self.board = [[' ' for \_ in range(7)] for \_ in range(6)]

        self.current\_player = None

        self.winning\_condition = 4

Purity: Pure

Totalization: Totalized

**print\_board(): Prints the game board to the console.**

  def print\_board(self):

        for row in self.board:

            print("| " + " | ".join(row) + " |")

            print("+---+---+---+---+---+---+---+")

Purity: Impure

Totalization: Not totalized

**is\_valid\_move(): Checks if a move is valid.**

  def is\_valid\_move(self, row, column):

        return row >= 0 and row < 6 and column >= 0 and column < 7 and self.board[row][column] == ' '

Purity: Pure

Totalization: Totalized

**make\_move(): Updates the game board with the player's move.**

  def make\_move(self, row, column, token):

        self.board[row][column] = token

Purity: Impure

Totalization: Totalized

**check\_winner(): Checks if the current player has won.**

    def check\_winner(self, token):

        # Check horizontal

        for row in range(6):

            for col in range(4):

                if all(self.board[row][col + i] == token for i in range(self.winning\_condition)):

                    return True

        # Check vertical

        for row in range(3):

            for col in range(7):

                if all(self.board[row + i][col] == token for i in range(self.winning\_condition)):

                    return True

        # Check diagonal (positive slope)

        for row in range(3):

            for col in range(4):

                if all(self.board[row + i][col + i] == token for i in range(self.winning\_condition)):

                    return True

        # Check diagonal (negative slope)

        for row in range(3):

            for col in range(3, 7):

                if all(self.board[row + i][col - i] == token for i in range(self.winning\_condition)):

                    return True

        return False

Purity: Pure

Totalization: Not totalized

**Player Class:**

class Player:

    def \_\_init\_\_(self, name, token):

        self.name = name

        self.token = token

The **\_\_init\_\_()** method in the Connect 4 game initializes player attributes, setting up the player's name and any other relevant properties. Let's delve into its characteristics in more detail:

**Functionality**: The primary purpose of the **\_\_init\_\_()** method is to initialize the player object with specific attributes. This includes assigning a name to the player, which serves as a unique identifier, and any other relevant properties that define the player's state or behavior within the game.

**Purity**: The **\_\_init\_\_()** method is classified as pure because it doesn't depend on external factors or produce side effects. It simply initializes the player object with the provided attributes and doesn't modify any external state.

**Totalization**: The **\_\_init\_\_()** method is totalized, meaning it covers all possible cases or scenarios for initializing player attributes. It ensures that all required attributes are properly set up, providing a consistent starting point for player objects.

**player\_move() Function:**

def player\_move(game, player):

    while True:

        try:

            row = int(input(f"{player.name}'s turn ({player.token}): Enter row (1-6): ")) - 1

            column = int(input(f"{player.name}'s turn ({player.token}): Enter column (1-7): ")) - 1

            if game.is\_valid\_move(row, column):

                game.make\_move(row, column, player.token)

                break

            else:

                print("Invalid move! Please choose an empty cell.")

        except ValueError:

            print("Invalid input! Please enter numbers.")

player\_move(): Handles the player's move input.

Purity: Impure

Totalization: Not totalized

**cpu\_move() Function:**

def cpu\_move(game, player):

    print(f"{player.name}'s turn ({player.token}):")

    while True:

        row = random.randint(0, 5)

        column = random.randint(0, 6)

        if game.is\_valid\_move(row, column):

            game.make\_move(row, column, player.token)

            break

The **cpu\_move()** function in the Connect 4 game generates a random move for the CPU player. However, it's important to note that this function is categorized as impure due to its reliance on randomness. Let's explore its characteristics further:

**Functionality**: The primary purpose of **cpu\_move()** is to select a column where the CPU player will drop its token. This selection is made randomly, simulating the CPU's move in the game. Since the CPU's strategy is not based on any specific logic or heuristic, the move is chosen at random.

**Purity**: The function is classified as impure because its behavior depends on external factors, specifically randomness. Each time **cpu\_move()** is called, it may produce a different result. This impurity makes the function less predictable and can complicate testing and debugging.

**Totalization**: The **cpu\_move()** function is not totalized, meaning it doesn't cover all possible cases or scenarios. It only generates a move without considering the current state of the game board or any strategic considerations. As a result, it may not always generate a valid move, especially when the game board is full or certain columns are already occupied.

**start\_game() Function:**

def start\_game():

    print("Welcome to Connect 4!")

    while True:

        try:

            num\_players = int(input("Enter number of players (1 or 2): "))

            if num\_players == 1 or num\_players == 2:

                break

            else:

                print("Invalid number of players! Please enter 1 or 2.")

        except ValueError:

            print("Invalid input! Please enter a number.")

    game = Connect4()

    player1\_name = input("Enter Player 1's name: ")

    player1\_token = input("Enter Player 1's token (X or O): ").upper()

    while player1\_token not in ['X', 'O']:

        player1\_token = input("Invalid token! Please enter X or O: ").upper()

    player1 = Player(player1\_name, player1\_token)

    player2\_token = 'X' if player1\_token == 'O' else 'O'

    if num\_players == 1:

        player2\_name = "CPU"

    else:

        player2\_name = input("Enter Player 2's name: ")

    player2 = Player(player2\_name, player2\_token)

    print(f"{player2.name}'s token is {player2.token}")

    game.current\_player = player1

    while True:

        print("\n" + "  ".join(str(i) for i in range(1, 8)))

        print("+---+---+---+---+---+---+---+")

        game.print\_board()

        if game.current\_player == player1:

            player\_move(game, player1)

        else:

            cpu\_move(game, player2) if num\_players == 1 else player\_move(game, player2)

        if game.check\_winner(game.current\_player.token):

            print("\n" + "  ".join(str(i) for i in range(1, 8)))

            print("+---+---+---+---+---+---+---+")

            game.print\_board()

            print(f"{game.current\_player.name} wins!")

            break

        if all(token != ' ' for row in game.board for token in row):

            print("\n" + "  ".join(str(i) for i in range(1, 8)))

            print("+---+---+---+---+---+---+---+")

            game.print\_board()

            print("It's a draw!")

            break

        game.current\_player = player2 if game.current\_player == player1 else player1

start\_game()

start\_game(): Initializes the game and controls its flow.

Purity: Impure

Totalization: Not totalized

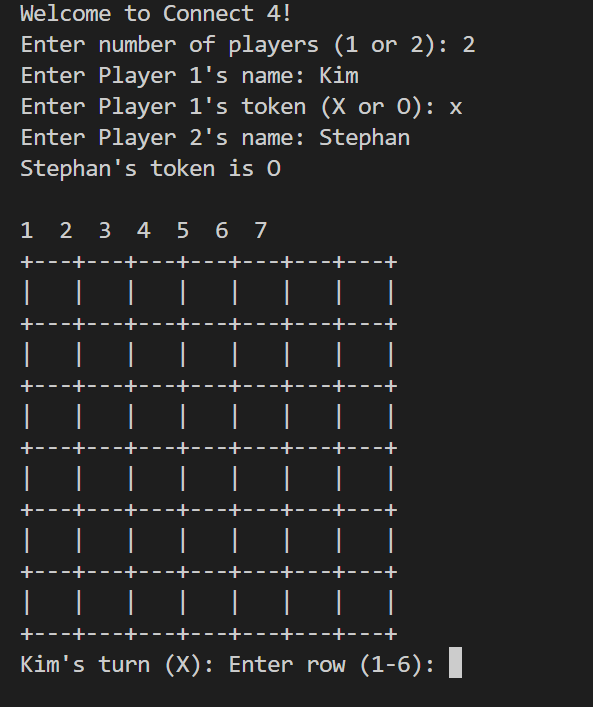
**Explanation/ Justification**:

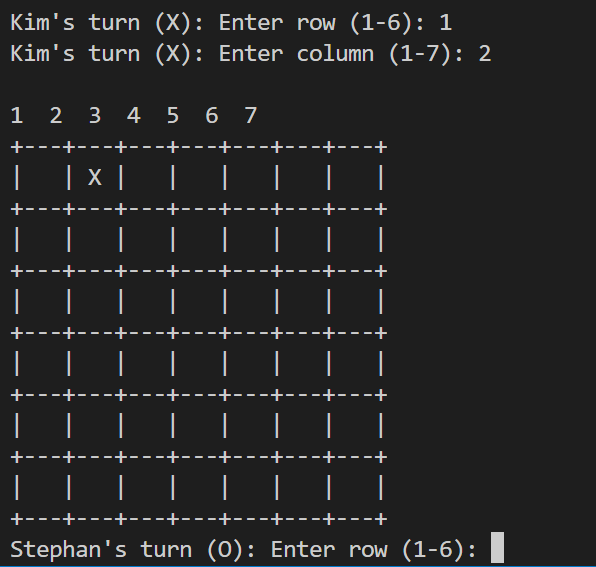
Pure functions only depend on their input parameters and do not modify any external state. In this implementation, \_\_init\_\_(), is\_valid\_move(), check\_winner(), and Player.\_\_init\_\_() are pure functions.

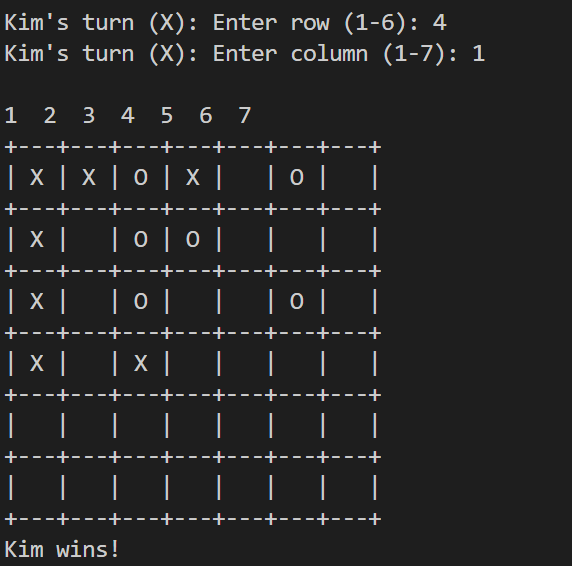
Impure functions either modify external state or have side effects. Functions like print\_board(), make\_move(), and player\_move() are impure because they either print to the console or modify the game state.

Totalized functions cover all possible input-value pairs. Pure functions, such as is\_valid\_move(), check\_winner(), and Player.\_\_init\_\_(), are totalized since they cover all possible valid inputs. Impure functions like print\_board(), make\_move(), player\_move(), and cpu\_move() are not totalized since they don't cover all possible inputs (for example, print\_board() only covers printing the current state of the board).

**Output**







**Testing**

Both manual and automated tests play crucial roles in ensuring that the software functions correctly and meets the specifications outlined in planning documents such as Gherkin scenarios.

Manual testing involves manually executing the software and observing its behavior to verify that it produces the expected outputs for specific user inputs. This process mimics how end-users would interact with the software. Manual tests are conducted step by step, where testers input data, interact with the user interface, and evaluate the system's responses.

For example, in a web application, manual testing might involve entering data into input fields, clicking buttons, navigating through different pages, and verifying that the displayed information matches expectations. Testers document their observations and any issues encountered during the manual testing process.

Manual testing is particularly useful for exploring edge cases, scenarios that are difficult to automate, and assessing the overall user experience. It provides insights into the system's usability, intuitiveness, and whether it aligns with user expectations.

However, manual testing can be time-consuming and prone to human error. Therefore, it's often complemented by automated testing. Automated tests are scripts or programs that execute predefined actions on the software and check whether the actual outcomes match the expected ones. This approach is faster, repeatable, and less error-prone than manual testing.

By combining manual and automated testing, software teams can ensure comprehensive test coverage, detect defects early in the development process, and deliver high-quality software that meets user needs and requirements.

**Manual Testing**

| **Test Case ID** | **1** | **Passed/Failed** |
| --- | --- | --- |

| **Feature** | **Start Game** |  |
| --- | --- | --- |
| Steps to Do | Enter number of players: 2 | "Welcome to Connect 4!" printed |
| Expected Output | "Enter Players name:" | Welcome message printed |
| Actual Result |  | |

| **Test Case ID** | **2** | **Passed/Failed** |
| --- | --- | --- |

| **Feature** | **Start Game** |  |
| --- | --- | --- |
| Steps to Do | Enter number of players: 3 | "Invalid number of players! Please enter 1 or 2." printed |
| Expected Output | - | Error message printed |
| Actual Result |  | |

| **Test Case ID** | **3** | **Passed/Failed** |
| --- | --- | --- |

| **Feature** | **Player Move** |  |
| --- | --- | --- |
| Steps to Do | Enter row: 4, Enter column: 2 | Token 'X' placed at (3, 1) on the board |
| Expected Output | Board with 'X' at (3, 1) | Updated board displayed |
| Actual Result |  | |

| **Test Case ID** | **4** | **Passed/Failed** |
| --- | --- | --- |

| **Feature** | **CPU Move** |  |
| --- | --- | --- |
| Steps to Do | CPU's turn | Random token placed on the board |
| Expected Output | Board with CPU's token placed | Updated board displayed |
| Actual Result |  | |

| **Test Case ID** | **5** | **Passed/Failed** |
| --- | --- | --- |

| **Feature** | **Check Winner** |  |
| --- | --- | --- |
| Steps to Do | Create winning condition | Check winner function returns True |
| Expected Output | "Player X wins!" or "Player O wins!" | Winner message displayed |
| Actual Result |  | |

| **Test Case ID** | **6** | **Passed** |
| --- | --- | --- |

| **Feature** | **Game continue until winner declare** |  |
| --- | --- | --- |
| Steps to Do | Continue playing | Check for a continue until win |
| Expected Output | "Game conitune " | Game continue |
| Actual Result |  | |

| **Test Case ID** | **7** | **Passed/Failed** |
| --- | --- | --- |

| **Feature** | **Check Valid Move** |  |
| --- | --- | --- |
| Steps to Do | Place token on an occupied cell | Check if move is valid |
| Expected Output | "Invalid move! Please choose an empty cell." | Error message displayed |
| Actual Result |  | |

| **Test Case ID** | **8** | **Passed/Failed** |
| --- | --- | --- |

| **Feature** | **Check Valid Move** |  |
| --- | --- | --- |
| Steps to Do | Enter row: -1, Enter column: 3 | Check if move is valid |
| Expected Output | "Invalid move! Please choose an empty cell." | Error message displayed |
| Actual Result |  | |

| **Test Case ID** | **9** | **Passed/Failed** |
| --- | --- | --- |

| **Feature** | **Check Valid Move** |  |
| --- | --- | --- |
| Steps to Do | Enter row: 4, Enter column: 8 | Check if move is valid |
| Expected Output | "Invalid move! Please choose an empty cell." | Error message displayed |
| Actual Result |  | |

| **Test Case ID** | **10** | **Passed/Failed** |
| --- | --- | --- |

| **Feature** | **Check Valid Move** |  |
| --- | --- | --- |
| Steps to Do | Enter row: a, Enter column: b | Check if move is valid |
| Expected Output | "Invalid input! Please enter numbers." | Error message displayed |
| Actual Result |  | |

| **Test Case ID** | **11** | **Passed/Failed** |
| --- | --- | --- |

| **Feature** | **Check Valid Move** |  |
| --- | --- | --- |
| Steps to Do | Enter row: 3.5, Enter column: 2.5 | Check if move is valid |
| Expected Output | "Invalid input! Please enter numbers." | Error message displayed |
| Actual Result |  | |

**Automated testing:**

The Python unittest library is highly regarded for automated testing, particularly in projects like Farkle. Being part of the standard Python library, it's easily accessible to all Python developers. Unittest provides a robust testing framework that enables developers to create well-structured and organized test cases.

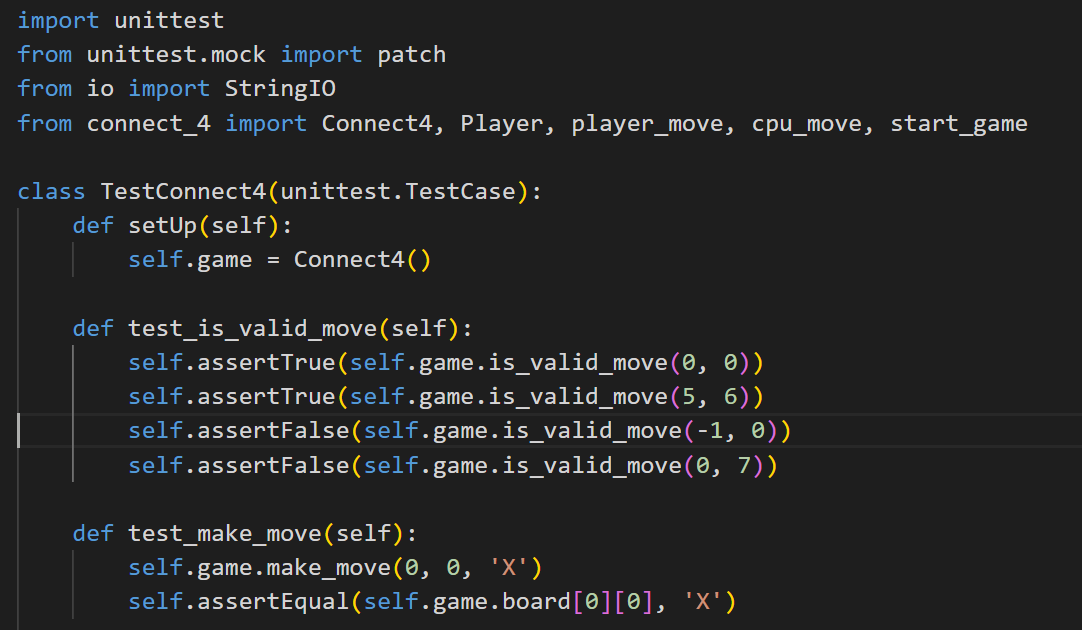
One of the key benefits of unittest is its simplicity and user-friendly interface. It allows developers to define test cases, run tests, and make assertions about code behavior with ease. Test cases are organized into classes, with each test method representing a specific scenario. This structured approach ensures clarity and maintainability of the test suite, even as the project grows.

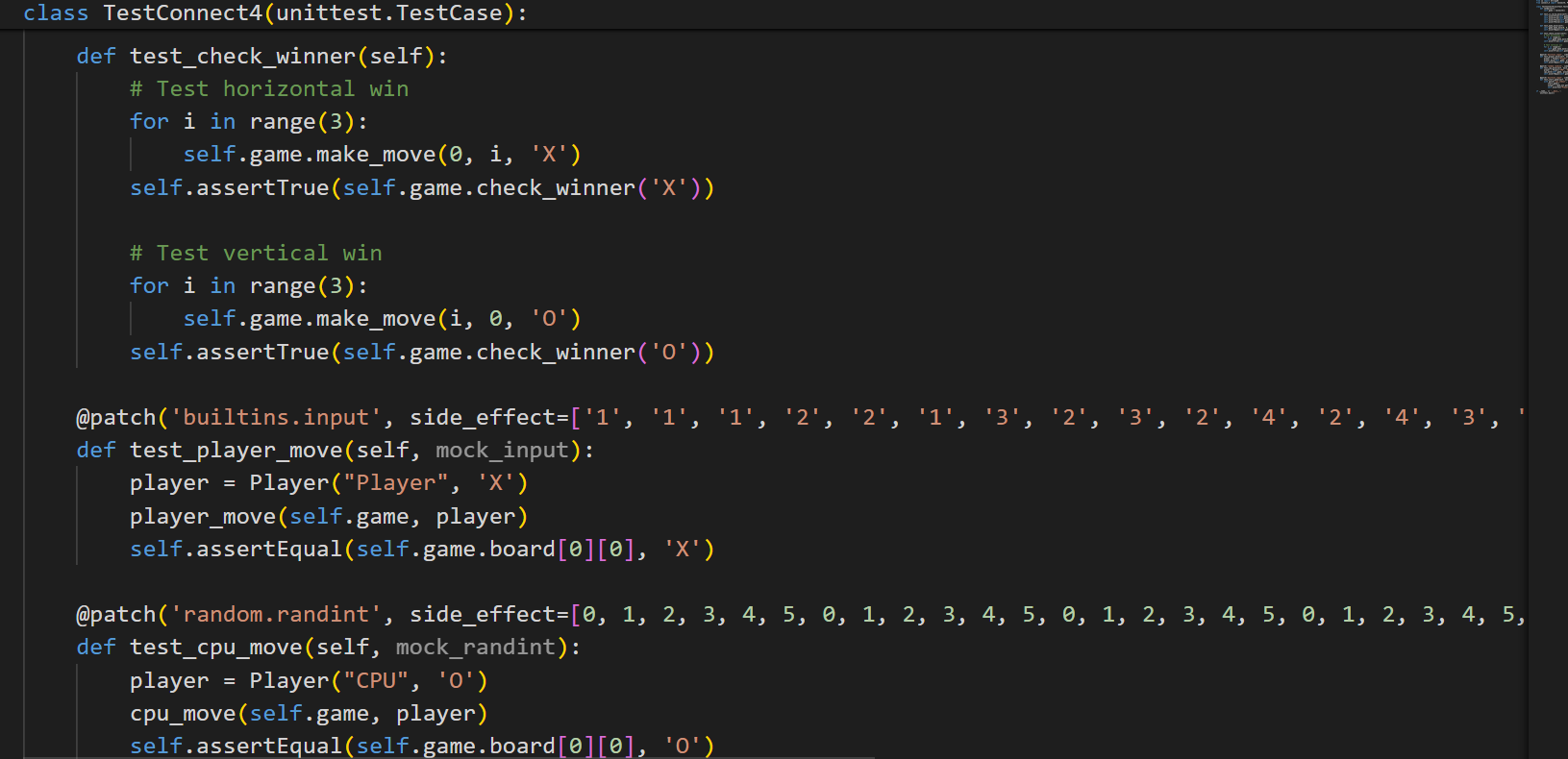
Unittest offers a variety of assertion methods for code verification, including equality, inequality, truthiness, and containment checks. These assertions help developers express their expectations about code behavior and ensure it functions as intended.

Moreover, unittest supports test fixtures, allowing developers to set up the testing environment before each test and clean up afterward. This ensures test isolation and independence, crucial for accurate testing.

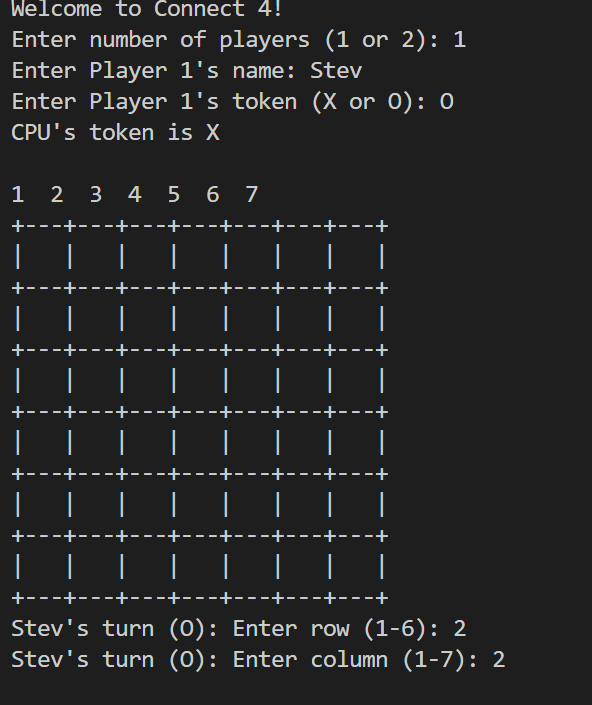
Test discovery is another valuable feature of unittest, automatically finding and running all test cases within a project. This simplifies running tests across the entire codebase, particularly in large projects with many test cases spread across multiple files.

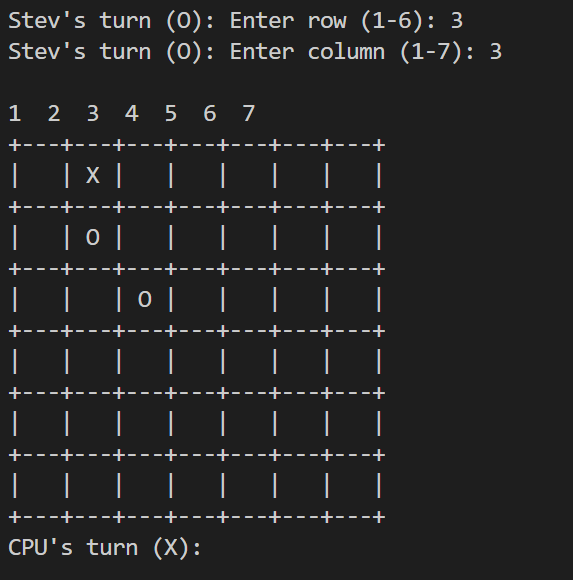
**Setup for testing**

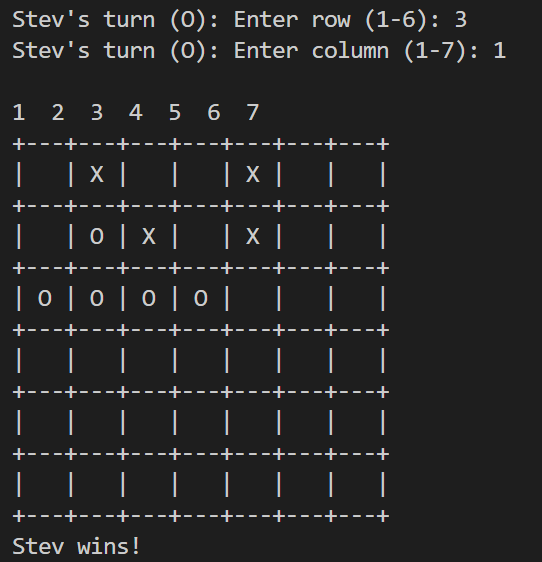




Running the code and output of tests:



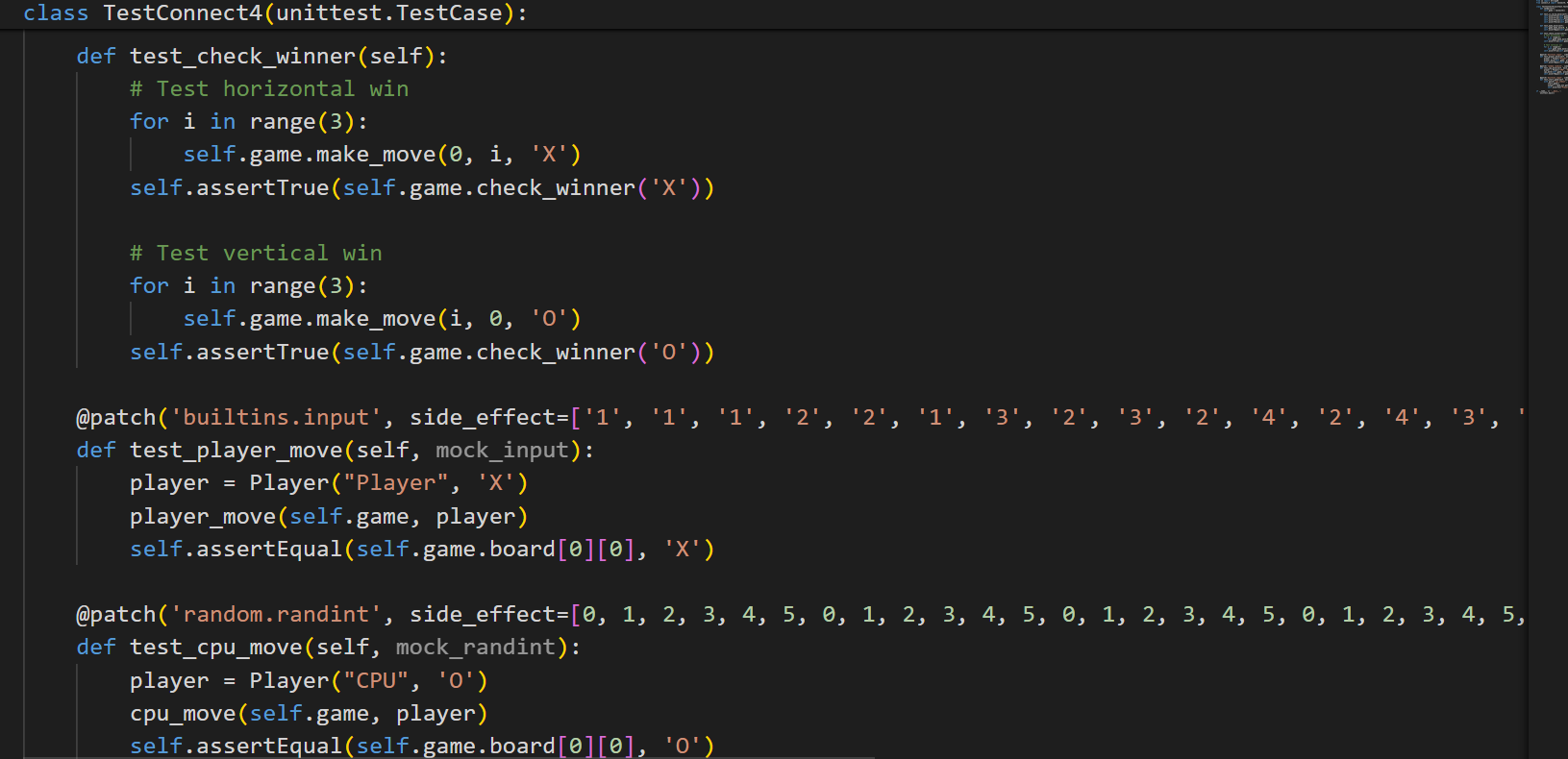


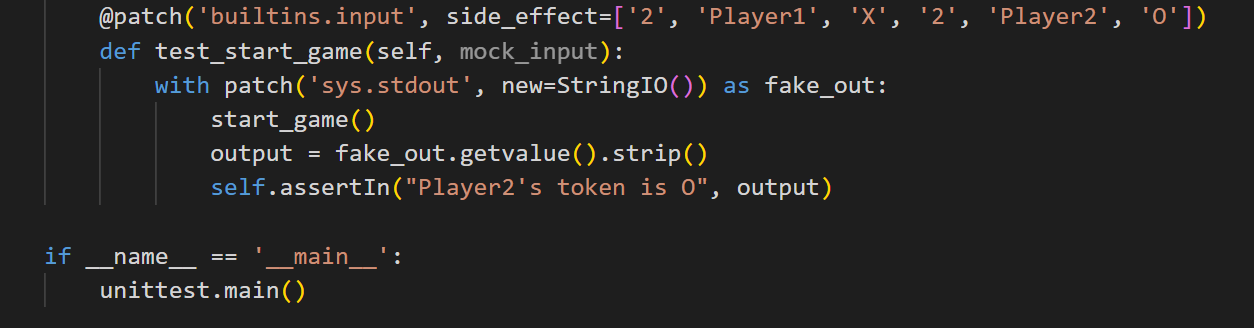


Game is stopped when user quit it.

**Appendix A**

**Full automated testing code:**





T4 Git version control discussion

Git commits serve as checkpoints in a project's history, capturing the state of the codebase at specific moments. These commits are essential for the Farkle project, providing developers with the ability to roll back to previous versions if necessary and maintaining a clear record of changes. This functionality enables quick recovery from errors or unintended modifications and facilitates tracking changes over time, ensuring the project's stability and progress.

Pushing commits to the Git repository ensures that changes made on a local machine are shared with others, granting team members access to the latest version of the codebase. Similarly, pulling updates the local machine with changes made by others, ensuring everyone is working with the most recent version. For team collaboration on the Farkle project, pull requests play a vital role. They enable developers to propose changes, undergo peer review, and then merge approved changes into the main repository, ensuring code quality and stability through thorough examination before integration.

Furthermore, pull requests foster collaboration and communication among team members, allowing for discussions and feedback on proposed changes. This collaborative approach helps ensure that code changes align with project goals and coding standards. By leveraging pull requests, the Farkle project can benefit from the collective expertise of team members, leading to better code quality and more robust features..

**Case study**

For a case study of git, I study nodejs repositoryt to get idea about git and github.

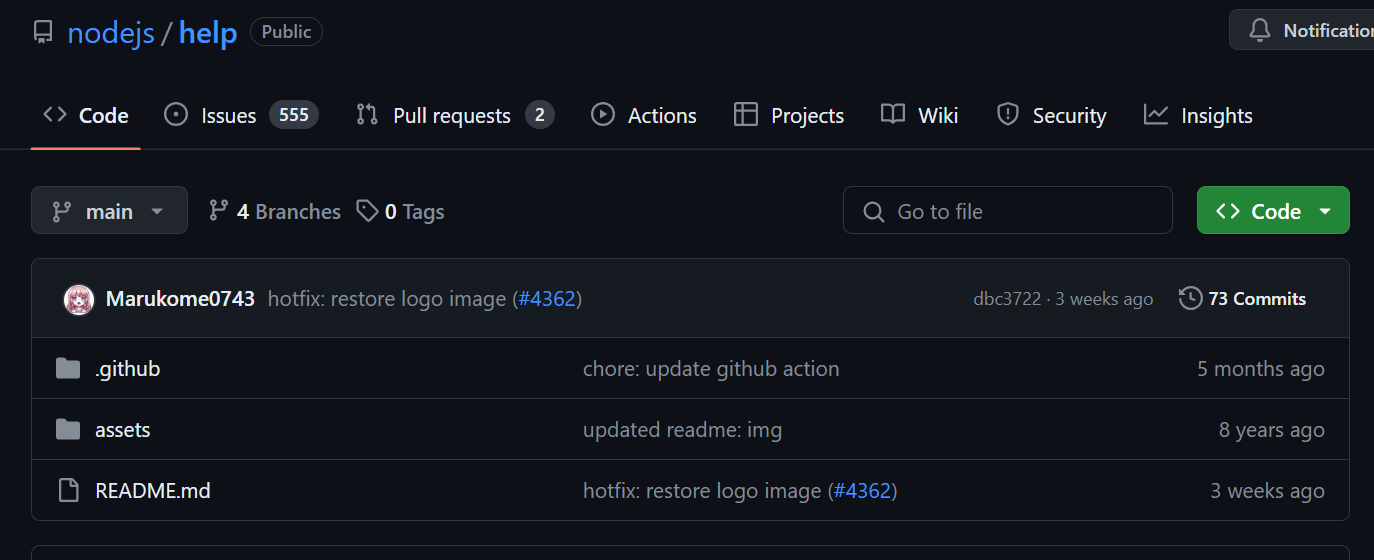
<https://github.com/nodejs/help>

Accessing the commit history for kislyuk on GitHub is straightforward. When navigating through the project's repository, you can easily find the commit history by clicking on the commits symbol, which is typically represented by a clock or a series of dots. This symbol is located at the top of the repository's page, next to other navigation options like branches, pull requests, and releases. In the navigation bar of the repository, the commits symbol is underlined in red in Figure 1.

Upon clicking on the commits symbol, you are directed to a page that displays a chronological list of all the commits made to the repository. Each commit entry includes details such as the commit message, the author, the timestamp, and a unique identifier (commit hash). This information provides a comprehensive overview of the changes made to the project over time.

Furthermore, GitHub provides various options for filtering and exploring the commit history. You can search for specific commits using keywords or filter commits by author, date range, or branch. Additionally, GitHub offers visualization tools, such as graphs and charts, to visualize the commit history in different ways, allowing for a deeper understanding of the project's development timeline.

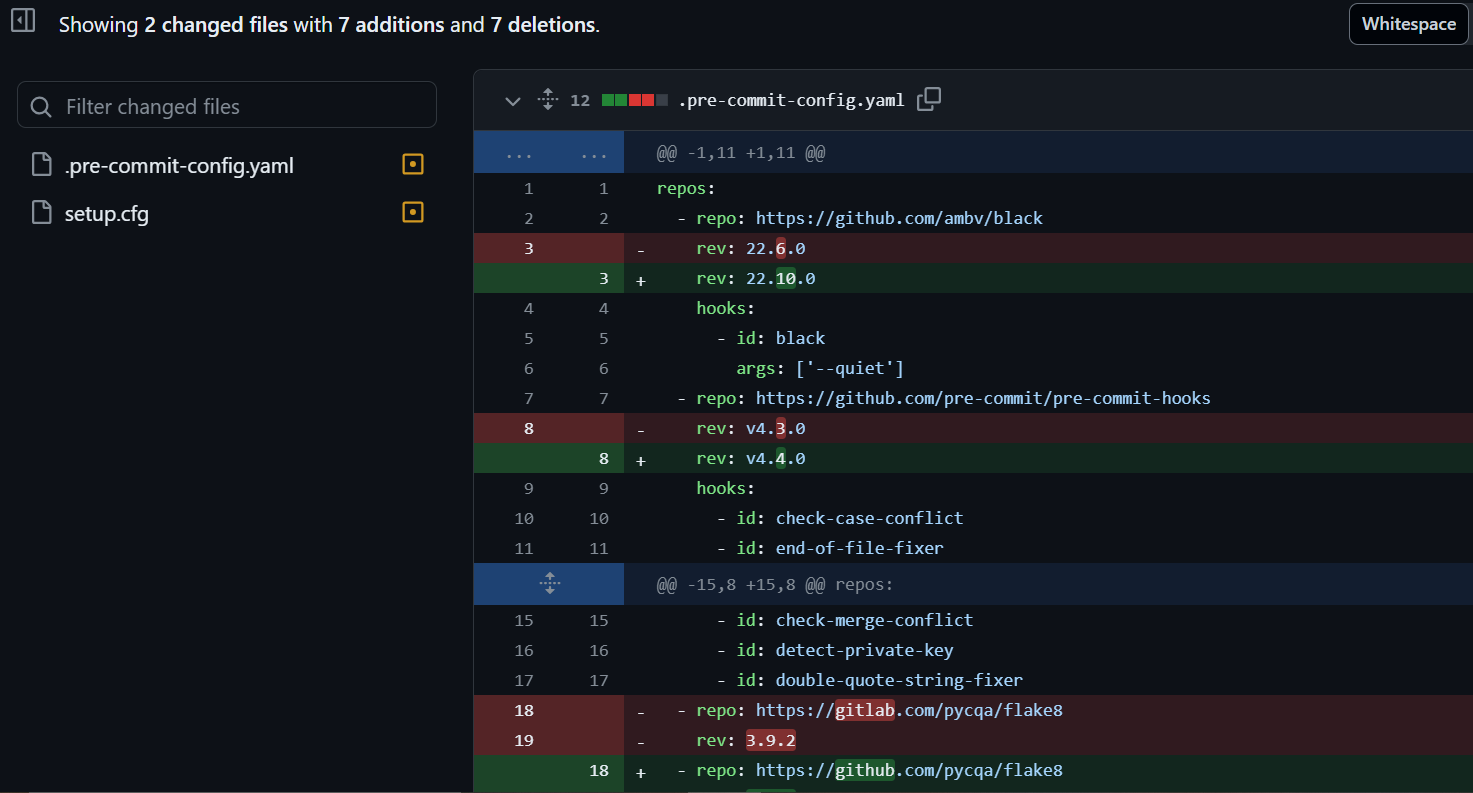
The commit history is an invaluable resource for tracking changes, understanding the evolution of the project, and identifying when and why specific modifications were made. It serves as a detailed log of the project's development journey, enabling developers to review past changes, collaborate effectively, and maintain code quality. By leveraging the commit history, contributors to the kislyuk project can gain insights into the codebase's evolution and make informed decisions about future development efforts..

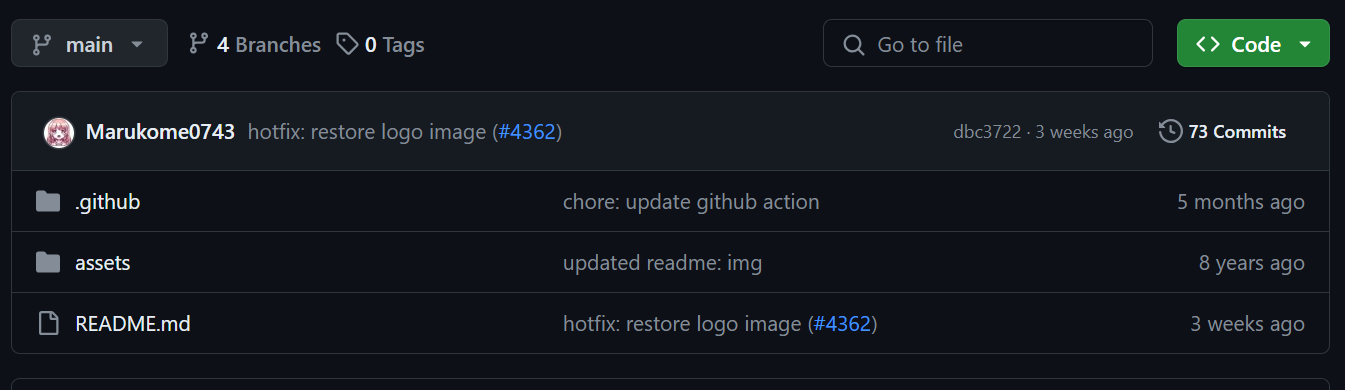


Commits on GitHub are presented in chronological order, with the most recent commit displayed at the top of the list. This arrangement allows users to quickly identify the latest changes made to the repository. As you scroll down the commit history page or select the "older" option, you can navigate through earlier commits in the repository. Each commit entry provides valuable information about the changes introduced in that particular commit, including the commit message, author, timestamp, and a unique commit hash.

GitHub highlights changes in the project code using red and green markers, providing a visual indication of modifications. When viewing a commit, additions to the code are highlighted in green, while deletions are highlighted in red. This color scheme makes it easy to discern the exact changes made to the codebase in each commit, helping developers understand the scope and impact of the modifications. The green highlights indicate lines of code that have been added or modified since the previous commit. Conversely, the red highlights denote lines of code that have been removed or altered.

This color-coded approach enhances the readability of the commit history, making it easier for developers to track changes and understand the evolution of the project over time. It provides valuable insights into the development process and facilitates collaboration among team members by clearly highlighting the contributions made by each commit. Through the use of red and green highlights, GitHub's commit history offers a visual representation of the project's code changes, enabling effective code review and version control.



The total number of branches or folders in a repository is displayed as the count of branches or directories present within the repository. This count includes both the main branch (typically named "master" or "main") and any other branches created during development. Additionally, if the repository contains subdirectories, each subdirectory is counted as a separate folder within the repository. 

**Refrences:**

The benefits and need of documentation of code:

<https://swiftspeedappcreator.com/appblog/importance-of-documentation-in-software-maintenance/>Accessed [4/05/2024]

Text connect 4 game guide:

<https://en.wikipedia.org/wiki/Connect_Four> Accessed [04/05/2024]

Git guid sample:

<https://github.com/git-guides/git-clone> Accessed [29/04/2024]