Comprehensive Exercise Report

Team **Alpha** of Section **007**

### Team members: Daniel Hudhra, Denisa Kercanaj, Glears Canaj, Chidiebube Mbakogu

NOTE: You will replace all placeholders that are given in <<>>

[**Requirements/Analysis**](#_heading=h.gjdgxs)2

[Journal](#_heading=h.30j0zll) 2

[Software Requirements](#_heading=h.1fob9te) 4

[**Black-Box Testing**](#_heading=h.3znysh7)4

[Journal](#_heading=h.2et92p0) 4

[Black-box Test Cases](#_heading=h.tyjcwt) 5

[**Design**](#_heading=h.3dy6vkm)6

[Journal](#_heading=h.1t3h5sf) 6

[Software Design](#_heading=h.4d34og8) 7

[**Implementation**](#_heading=h.2s8eyo1)8

[Journal](#_heading=h.17dp8vu) 8

[Implementation Details](#_heading=h.3rdcrjn) 9

[**Testing**](#_heading=h.26in1rg)10

[Journal](#_heading=h.lnxbz9) 10

[Testing Details](#_heading=h.35nkun2) 11

[**Presentation**](#_heading=h.1ksv4uv)12

[Preparation](#_heading=h.44sinio) 12

#### Grading Rubric Error! Bookmark not defined.

# Requirements/Analysis

Week 2

## Journal

The following prompts are meant to aid your thought process as you complete the requirements/analysis portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

* After reading the client’s brief (possibly incomplete description), write one sentence that describes the project (expected software) and list the already known requirements.
  + After reading the client’s brief, the project involves developing software for a digital version of Connect Known requirements include:
* Ability to display the game board and pieces
* Support for two players to take turns placing their pieces
* Winning condition detection
* User interface for player interactions
* After reading the client’s brief (possibly incomplete description), what questions do you have for the client? Are there any pieces that are unclear? After you have a list of questions, raise your hand and ask the client (your instructor) the questions; make sure to document his/her answers.
  + Questions for the client:
* Are there any specific rules or variations of Connect 4 we should consider?
* Should the game have options for different board sizes or difficulty levels?
* Will there be any additional features like a replay option or AI opponents?

#### Answers from instructor:

1. Standard Connect 4 rules apply.
2. Stick to the standard board size.
3. No additional features required for now.

* Does the project cover topics you are unfamiliar with? If so, look up the topics and list your references.

AI algorithms for game playing, graphical user interface development.

* Describe the users of this software (e.g., small child, high school teacher who is taking attendance).

Users will be players interested in playing Connect 4 against each other.

* Describe how each user would interact with the software

Users will interact through a graphical user interface where they can select columns to drop their pieces, observe the game board, and receive notifications about game outcomes.

* What features must the software have? What should the users be able to do?

Game board display Player turn management

Winning condition detection

#### Requirements/Analysis Week 2

## Software Requirements

<<Use your notes from above to complete this section of the formal documentation by writing a detailed description of the project, including a paragraph overview of the project followed by a list of requirements (see lecture for format of requirements). You may also choose to include user stories.>>

*The Connect 4 digital version project aims to develop software that provides an engaging and intuitive platform*

*for players to enjoy the classic game of Connect 4 in a digital format. The software will feature a graphical user*

*interface (GUI) where players can interact seamlessly, taking turns placing their colored discs on the game*

*board. The primary goal of the software is to faithfully replicate the rules and mechanics of the traditional*

*Connect 4 game while offering a user-friendly experience.*

**Software Requirements:**

Game Board Display:

• The software shall display a visually appealing game board consisting of vertical columns and rows

where players can drop their discs.

• The game board shall accurately represent the current state of the game, including the positions of the

discs placed by each player.

Player Turn Management:

• The software shall support two players, allowing them to take turns placing their colored discs on the

game board.

• The system shall enforce turn-based gameplay, ensuring that each player has the opportunity to make

a move before switching to the next player.

Winning Condition Detection:

• The software shall detect and identify winning conditions, such as when a player achieves four of their

colored discs in a row vertically, horizontally, or diagonally.

• Upon detecting a winning condition, the software shall declare the corresponding player as the winner

and display a winning animation or message.

User Interface:

• The software shall provide a user-friendly graphical user interface (GUI) for player interactions.

• The GUI shall allow players to select columns to drop their discs, observe the game board, and receive

notifications about game outcomes such as wins, ties, or invalid moves.

User Stories:

• As a player, I want to see a visually appealing game board displayed on the screen, so I can easily

understand the game state.

• As a player, I want to take turns with my opponent, so we can compete fairly in the game.

• As a player, I want the software to detect when I achieve four discs in a row, so I can be declared the

winner promptly.

• As a player, I want the GUI to be intuitive and easy to navigate, so I can focus on playing the game

without confusion.

• As a player, I want the interface to be interactive.

• Also, I do not want the game to be monotonous, but the game should, for example, train the AI to make

good decisions, preferably using Minimax and Alpha-Beta Pruning algorithms.

# Black-Box Testing

Instructions: Week 4

## Journal

***Remember:*** Black box tests should only be based on your requirements and should work independent of design.

The following prompts are meant to aid your thought process as you complete the black box testing portion of this exercise. Please review your list of requirements and respond to each of the prompts below. Feel free to add additional notes.

* What does input for the software look like (e.g., what type of data, how many pieces of data)?
* The input for our software consists of the player's selection of a column to drop a disc.
* This input can be represented as an **integer value** corresponding to the column number.
* This input will be provided by the player clicking on a column in the graphical user interface.
* What does output for the software look like (e.g., what type of data, how many pieces of data)?
* **In the context of our “Connect 4” game,** the output for the software includes the updated grid displaying the discs after each player's move, along with notifications of game events such as a player achieving four in a row, a tie game, or prompts for invalid moves.
* What equivalence classes can the input be broken into?
* In general, the input can be classified into the equivalence classes of valid column numbers (1 to 7) and invalid column numbers (less than 1 or greater than 7).
* Therefore, in our GUI-based game, the input can be classified into the equivalence classes of valid column selections (clicks within the range of columns displayed on the GUI) and invalid column selections (clicks outside the range of columns).
* What boundary values exist for the input?
* Boundary values include the minimum valid column number (1), maximum valid column number (7), and values outside this range (0 and 8).
* In our GUI-based game, the boundary values include the minimum valid column number (the leftmost column displayed on the GUI) and the maximum valid column number (the rightmost column displayed on the GUI).
* Are there other cases that must be tested to test all requirements?
* Apart from testing valid and invalid column numbers, testing for winning conditions (**horizontal, vertical, diagonal**) and tie game scenarios is necessary to cover all requirements.
* Other notes:
* It is essential for us to ensure that the game interface functions smoothly, allowing players to

*select columns* and *displaying the updated grid accurately after each move*.

* Moreover, it is fundamental to ensure that the GUI interface functions smoothly.

## Black-box Test Cases

Use your notes from above to complete the black-box test plan section of the formal documentation by writing black box test cases (other than actual results since no program currently exists). Remember to test each equivalence class, boundary value, and requirement.

| **Test ID** | **Description** | **Expected Results** | **Actual Results** |
| --- | --- | --- | --- |
| T1 | Testing a valid column selection within the range (e.g., column 3). | The disc should be visually placed in the lowest available position in the selected column on the grid. | N/A (since no program currently exists). |
| T2 | Testing an invalid column selection (e.g., clicking on a space outside of the grid). | The GUI should not register the click as a valid move, and the player should not be able to place a disc. | N/A (since no program currently exists). |
| T3 | Testing the winning condition with four discs aligned horizontally. | The game should declare the player who achieved four in a row as the winner, and a winning animation or message should be displayed. | N/A (since no program currently exists). |
| T4 | Testing the winning condition with four discs aligned vertically. | The game should declare the player who achieved four in a row as the winner, and a winning animation or message should be displayed. | N/A (since no program currently exists). |
| T5 | Testing the winning condition with four discs aligned diagonally. | The game should declare the player who achieved four in a row as the winner, and a winning animation or message should be displayed. | N/A (since no program currently exists). |
| T6 | Testing for a tie game scenario where the grid is filled without any player achieving four in a row. | The game should declare a tie, and a tie game message should be displayed. | N/A (since no program currently exists). |

# Design

Instructions: Week 6

## Journal

***Remember:*** You still will not be writing code at this point in the process.

The following prompts are meant to aid your thought process as you complete the design portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

* List the nouns from your requirements/analysis documentation.
* Software
* Connect 4
* Project
* Platform
* Players
* Game Board
* Discs
* Turn
* Game State
* Winner
* Animation
* Message
* object
* Connect4GUI

These nouns represent various elements and concepts related to the Connect 4 digital version project described in the documentation.

* Which nouns potentially may represent a class in your design?
* Connect4GUI so that we can create an instance of the Connect 4 game GUI.
* Which nouns potentially may represent attributes/fields in your design? Also list the class each attribute/field would be a part of.
* Regarding the connect4.Connect4GUI class, as fields (attributes), we can use canvas, buttons, root, turn, game\_over, info\_label, board etc.
* Now that you have a list of possible classes, consider different design options (***lists of classes and attributes***) along with the pros and cons of each. We often do not come up with the best design on our first attempt. Also consider whether any needed classes are missing. These two design options should not be GUI vs. non-GUI; instead you need to include the classes and attributes for each design. Reminder: Each design must include at least two classes that define object types.
* List at least two design options with pros and cons of each

**Design Option 1:** Simple Board and Game Classes

This design features two main classes: Board and Game. The Board class has attributes like grid, ROW\_COUNT, and COLUMN\_COUNT, with methods for initializing the board, dropping a piece, checking valid locations, finding the next open row, and checking for a winning move. The Game class includes attributes such as board, current\_turn, game\_over, PLAYER\_ONE, PLAYER\_TWO, and PLAYER\_COLORS. It has methods for initialization, switching turns, checking if the game is over, and starting a new game.

Pros: This design is straightforward, making it easy to understand and implement. It effectively separates the concerns, with the Board class focusing on board-specific logic and the Game class managing the game state and player turns.

Cons: The design is limited in flexibility, potentially becoming difficult to extend for additional features like an AI opponent or advanced win conditions. There's also tight coupling, as the Game class is heavily reliant on the Board class's implementation.

**Design Option 2:** Modular with Separate Player Class

This design introduces three classes: **Board**, **Player**, and **Game**. The Board class remains similar to the first design with attributes and methods for managing the game board. The Player class has attributes like name, color, and piece, with an initialization method. The Game class now includes a list of Player instances along with attributes for the board and current player turn, and methods for game management such as initializing, switching turns, checking for game over, and starting a new game.

Pros: This design is more modular and flexible, allowing for easier extension and maintenance. The separation of the Player class makes it simpler to add features related to player management, such as custom player settings or AI players.

Cons: The increased number of classes adds complexity, which might be unnecessary for a simple Connect 4 game. The design also requires careful coordination between classes, potentially leading to more intricate interdependencies.

* Which design do you plan to use? Explain why you have chosen this design.

For now, we choose design option 1: **Simple Board and Game Classes**.

**Explanation of our choice:**

Class Structure:

Board Logic:

Functions like create\_board, drop\_piece, is\_valid\_location, get\_next\_open\_row, and winning\_move manage board operations.

Game Management: The Connect4GUI class handles the game state, player turns, and GUI updates, integrating both game logic and interface control.

Responsibilities:

The Connect4GUI class initializes the board, manages player turns, checks for game over, and updates the GUI.

Board-related functions are directly called within the methods of the Connect4GUI class.

**Reason for Choosing this Design:**

This design is *easy to* ***understand and implement***, keeping the game logic in functions and combining game state management with GUI in a single class.

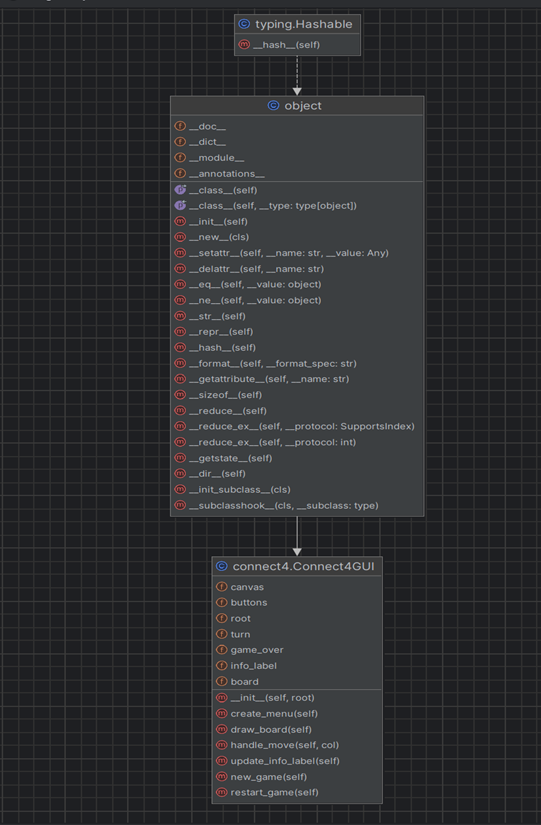
With fewer classes, this design is less complex and more accessible, especially for beginners.

It maintains a clear division between board logic (***via functions***) and game state management (in Connect4GUI), ensuring organized and maintainable code.

* List the verbs from your requirements/analysis documentation.
* **Create:** Used to instantiate objects or data structures, such as creating the game board or initializing player pieces.
* **Import:** Used to bring in external modules or libraries required for the program, like importing numpy for matrix operations or tkinter for GUI development.
* **Initialize:** Used to set up initial states or values, like initializing the game board with empty cells or setting player turns.
* **Drop:** Used to place a game piece onto the board at a specified location, representing a player's move in Connect 4.
* **Check:** Used to verify conditions or states, such as checking for valid locations to drop a piece or checking for a winning condition.
* **Get:** Used to retrieve information or data, such as getting the next available row in a column to drop a piece or retrieving player input.
* **Draw:** Used to render graphics or visual elements, like drawing the game board or updating the GUI with player pieces.
* **Handle:** Used to manage or process events or actions, such as handling player moves or responding to user interactions in the GUI.
* **Update:** Used to modify or refresh states or displays, such as updating the game board after a move or updating the GUI with current game state information.
* **Restart:** Used to reset the game to its initial state, allowing players to start a new game without closing the program (of course the board will become empty again).
* **Start:** Used to initiate or begin a process, like starting the main game loop or launching the GUI interface.
* Which verbs potentially may represent a method in your design? Also list the class each method would be part of.
* For example, a method of the Board class is **Drop** that is responsible for placing a game piece onto the board at a specified location.
* **Check:** Method of the Board class, responsible for verifying conditions such as checking for valid locations to drop a piece or checking for a winning condition.
* **Get:** Method of the Board class, responsible for retrieving information or data, such as getting the next available row in a column to drop a piece.
* **Draw:** Method of the Connect4GUI class, responsible for rendering graphics or visual elements, like drawing the game board or updating the GUI with player pieces.
* **Handle:** Method of the Connect4GUI class, responsible for managing or processing events or actions, such as handling player moves or responding to user interactions in the GUI.
* **Update:** Method of the Connect4GUI class, responsible for modifying or refreshing states or displays, such as updating the game board after a move or updating the GUI with current game state information.
* **Restart:** Method of the Connect4GUI class, responsible for resetting the game to its initial state, allowing players to start a new game without closing the program.
* **Start:** Method of the Connect4GUI class, responsible for initiating or beginning a process, like starting the main game loop or launching the GUI interface.
* Other notes:
* <<Insert notes>>

## Software Design

<<Use your notes from above to complete this section of the formal documentation by planning the classes, methods, and fields that will used in the software. Your design should include UML class diagrams along with method headers. ***Prior to starting the formal documentation, you should show your answers to the above prompts to your instructor.****>>*

**

# Implementation

Instructions: Week 8

## Journal

The following prompts are meant to aid your thought process as you complete the implementation portion of this exercise. Please respond to each of the prompt below and feel free to add additional notes.

* What programming concepts from the course will you need to implement your design? Briefly explain how each will be used during implementation.
* <<Insert answer>>

During our implementation process, we used plenty of our programming knowledge within the Python file:

**Methods:**

These represent actions or behaviors that objects of a class can perform. For instance, in our code, methods such as ***drop\_piece()*** and ***handle\_move()*** are used to execute actions like placing a game piece on the board and managing player moves, respectively. These methods belong to classes like Connect4GUI and are essential for controlling the game's flow and behavior.

**Classes:**

Classes are blueprints for creating objects that encapsulate data and behavior. In our code, ***Connect4GUI*** is a class responsible for managing the game's graphical user interface (GUI), while tk.Button and tk.Label represent classes provided by the tkinter library for creating GUI elements like buttons and labels. Each class defines attributes and methods that define the behavior and specific properties of its instances.

**Variables:**

These are used to store and manipulate data. In the provided code, variables like ROW\_COUNT, COLUMN\_COUNT, and EMPTY are constants used to define the dimensions of the game board and represent empty cells. Additionally, variables like self.board and self.turn are instance variables within the Connect4GUI class, storing the current state of the game.

**Control Structures:**

These structures determine the flow of execution in a program. For example, it includes loops and conditional statements. In our code, loops are used in functions like ***winning\_move()*** to iterate over the game board and check for winning conditions. Conditional statements, such as if and else, are utilized throughout the code to make decisions based on certain conditions, such as whether a move is valid or if a player has won.

**Libraries:**

Libraries, also known as modules or packages, provide pre-written code for specific functionalities. In our code, the ***numpy*** library is imported to create and manipulate arrays representing the game board, while the tkinter library is imported to create the GUI. These libraries offer a range of functions and classes that simplify development by providing ready-made solutions for common tasks.

**Event Handling:**

This concept involves responding to user interactions, such as mouse clicks or button presses. In our code, event handling is crucial for capturing player moves and updating the GUI accordingly. Methods like ***handle\_move()*** are triggered in response to user actions, allowing the game to progress based on player input.

* Other notes:
* <<Insert notes>>

## Implementation Details

<<Use your notes from above to write code and complete this section of the formal documentation with a README for the user that explains how he/she will interact with the system.>>

Everything required is uploaded in the link down below:

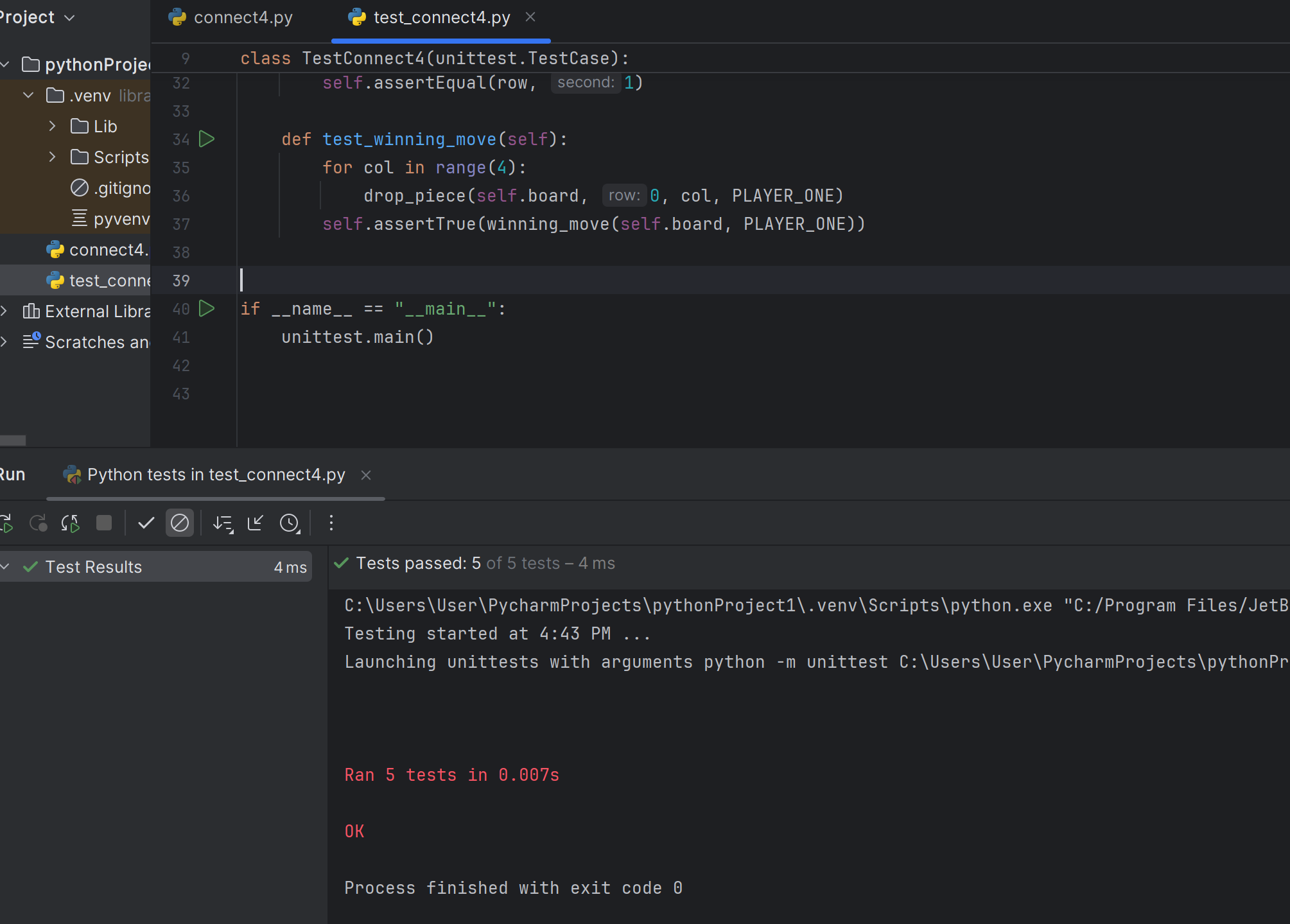
<https://github.com/danielhudhra19/pythonProject1>

# Testing

Instructions: Week 10

## Journal

The following prompts are meant to aid your thought process as you complete the testing portion of this exercise. Please respond to each of the prompts below and feel free to add additional notes.

* Have you changed any requirements since you completed the black box test plan? If so, list changes below and update your black-box test plan appropriately.
* The functionality described in the initial black-box testing plan still aligns with the implemented code.
* However, to ensure comprehensive testing coverage, we may need to consider additional scenarios such as:
* Testing the GUI elements' responsiveness and functionality, including button clicks and menu interactions.
* Verifying the behavior of the game when starting a new game or restarting the current game.
* Ensuring that the game accurately detects winning conditions and displays the appropriate message.
* Checking for proper error handling and notification display when invalid moves are attempted.
* List the classes of your implementation. For each class, list equivalence classes, boundary values, and paths through code that you should test.
* **Connect4GUI Class:**
* Equivalence Classes:
* Valid column selections: Clicks within the range of columns displayed on the GUI.
* Invalid column selections: Clicks outside the range of columns.
* Boundary Values:
* Minimum valid column number: 0 (leftmost column displayed on the GUI).
* Maximum valid column number: 6 (rightmost column displayed on the GUI).
* Paths to Test:
* Verify that the buttons representing valid columns respond correctly when clicked.
* Test clicking on areas outside the range of column buttons to ensure proper error handling.
* Check the behavior of menu options such as "New Game" and "Restart Game" to ensure they reset the game state appropriately.
* Test the GUI responsiveness by interacting with it in various scenarios, including winning conditions and invalid moves.
* **Helper Functions:**
* Equivalence Classes:
* Valid and invalid board states: Empty cells vs. cells occupied by player pieces.
* Different player moves: Dropping pieces for both players.
* Winning and non-winning board configurations.
* Boundary Values:
* Minimum row and column counts: 6 rows, 7 columns.
* Maximum row and column counts: 6 rows, 7 columns.
* Empty and occupied cell states: 0 for empty, 1 or 2 for player pieces.
* Paths to Test:
* Test the drop\_piece function by dropping pieces in various columns and verifying the resulting board state.
* Verify that the is\_valid\_location function correctly identifies valid and invalid column selections.
* Test different winning configurations using the winning\_move function to ensure correct detection of winning conditions.
* Check that the create\_board function generates an empty board of the correct dimensions.
* Test edge cases such as completely filled boards and boards with winning configurations in different directions.
* Other notes:
* 

import unittest

import numpy as np

from connect4 import create\_board, drop\_piece, is\_valid\_location, get\_next\_open\_row, winning\_move, PLAYER\_ONE, \

PLAYER\_TWO, EMPTY

class TestConnect4(unittest.TestCase):

def setUp(self):

self.board = create\_board()

def test\_create\_board(self):

self.assertTrue((self.board == np.zeros((6, 7), dtype=int)).all())

def test\_drop\_piece(self):

drop\_piece(self.board, 0, 0, PLAYER\_ONE)

self.assertEqual(self.board[0][0], PLAYER\_ONE)

def test\_is\_valid\_location(self):

self.assertTrue(is\_valid\_location(self.board, 0))

for row in range(6):

drop\_piece(self.board, row, 0, PLAYER\_ONE)

self.assertFalse(is\_valid\_location(self.board, 0))

def test\_get\_next\_open\_row(self):

row = get\_next\_open\_row(self.board, 0)

self.assertEqual(row, 0)

drop\_piece(self.board, 0, 0, PLAYER\_ONE)

row = get\_next\_open\_row(self.board, 0)

self.assertEqual(row, 1)

def test\_winning\_move(self):

for col in range(4):

drop\_piece(self.board, 0, col, PLAYER\_ONE)

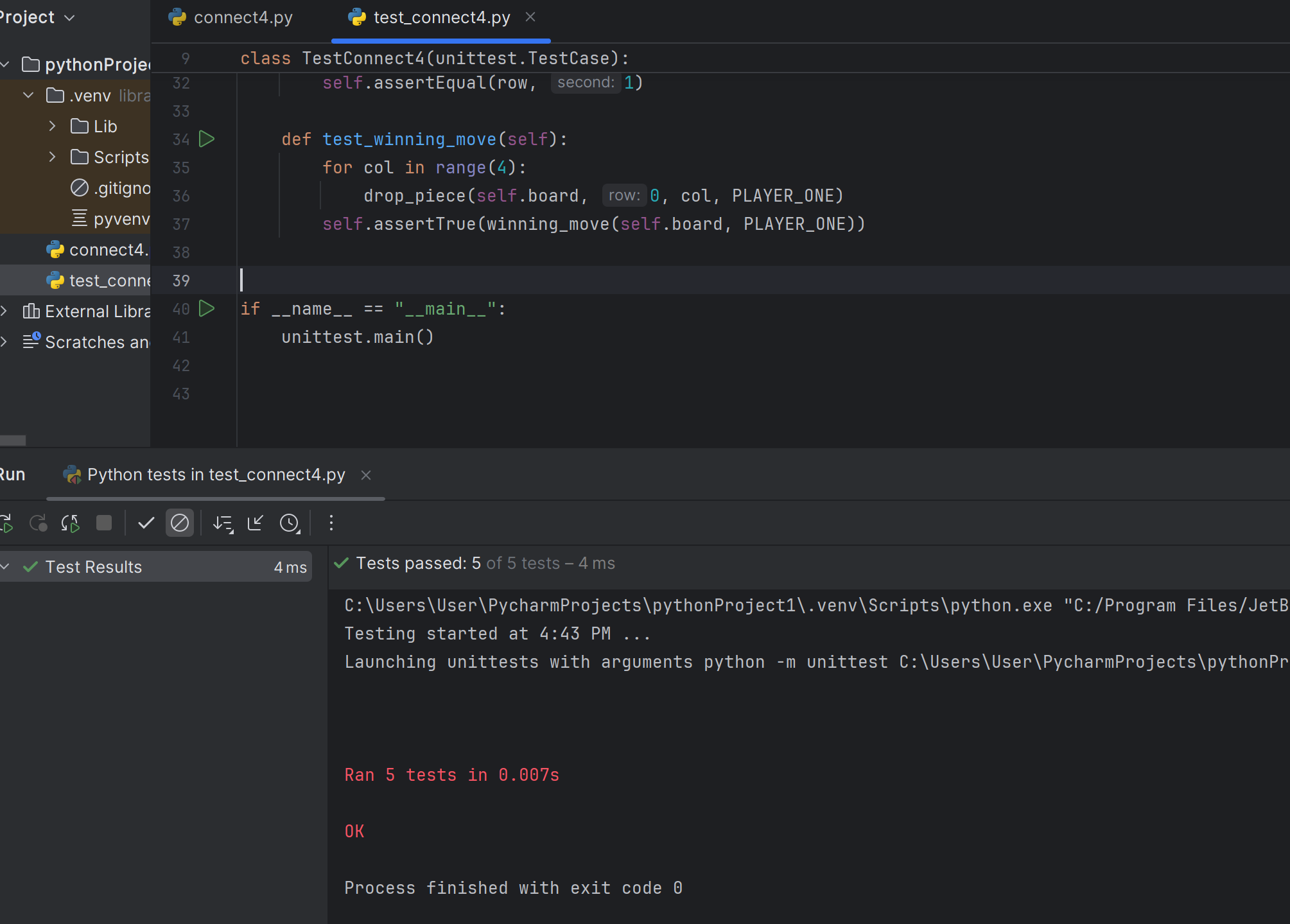
self.assertTrue(winning\_move(self.board, PLAYER\_ONE))

if \_\_name\_\_ == "\_\_main\_\_":

unittest.main()

## Testing Details

<<Use your notes from above to write your test programs and complete this section of the formal documentation by creating a list of your test programs along with descriptions of what they are testing. You will also complete the black-box test plan by running the program and filling in the Actual Results column.>>

* 

# Presentation

Instructions:Week 12

## Preparation

The following prompts are meant to aid your thought process as you complete the presentation portion of this exercise. It is recommended that you examine the previous sections of the journal and your reflections as you work on the presentation as it is likely that you have already answered some of the following prompts elsewhere. Please respond to each of the prompts below and feel free to add additional notes.

* Give a brief description of your final project
* Our final project is a graphical implementation of the classic Connect 4 game using Python, with the help of the **Tkinter** library for the graphical user interface (GUI) and **NumPy** for handling the game board's logic. The project allows two players to take turns dropping colored pieces (red and yellow) into a 6x7 grid, with the objective of getting four of their pieces in a row either horizontally, vertically, or diagonally.
* Describe your requirement assumptions/additions.
* **Requirement Assumptions:**
* Two-Player Local Mode:
* The game is designed for two players playing locally on the same machine. There is no network or AI opponent functionality.
* **Standard Connect 4 Board Dimensions:**
* The board is fixed at 6 rows by 7 columns, consistent with the traditional Connect 4 game.
* **Player Turns:**
* Players take turns to drop their pieces, with the game enforcing turn alternation.
* **Win Condition:**
* A player wins by aligning four pieces consecutively in a horizontal, vertical, or diagonal line.
* **Piece Dropping Mechanism:**
* Players select the column in which they wish to drop their piece, and the piece will occupy the lowest available row in that column.
* **Graphical User Interface (GUI):**
* The game uses Tkinter to create the GUI, which includes buttons for each column and a canvas to display the game board.
* **Game End Detection:**
* The game will display a message when a player wins and prevent further moves until the game is restarted or a new game is started.
* **Additions/Enhancements:**
* **Menu Options:**
* New Game: Starts a fresh game with an empty board.
* Restart Game: Resets the current game without exiting the application.
* Exit: Closes the application.
* Graphical Representation:
* The board is visualized on a Tkinter Canvas with colored circles representing player pieces (red for Player 1 and yellow for Player 2).
* **Turn Indicator:**
* A ***label*** that updates to indicate the current player's turn, switching between "Player 1's Turn (Red)" and "Player 2's Turn (Yellow)".
* **User Interaction:**
* Players click buttons corresponding to each column to drop their pieces.
* **Win Announcement:**
* A message box that appears to announce the winner once four consecutive pieces are detected.
* **Potential Future Enhancements:**
* **AI Opponent:**
* Implement an AI for single-player mode, allowing players to compete against the computer.
* **Improved Graphics and Animations:**
* Enhance visual elements with better graphics and animations for piece drops.
* **Player Customization:**
* Allow players to enter and display their names, enhancing personalization.
* **Undo Move Feature:**
* Implement a functionality to undo the last move made.
* **Save/Load Game:**
* Provide options to save the current game state and load it later, enabling players to pause and resume games.
* **Variable Board Sizes:**
* Allow players to select different board dimensions and adjust the game logic accordingly.
* **Sound Effects:**
* We can add audio feedback for actions like dropping pieces and winning moves to enhance user experience.
* Describe your design options and decision. How did you weigh the pros and cons of the different designs to make your decision?
* Design Options and Decisions:
* **GUI Framework:**
* Options: Tkinter, PyQt, Kivy
* *Decision: Tkinter*
* Pros: Built-in, simple, sufficient for project needs.
* Cons: Basic aesthetics.
* Board Representation:
* **Options: 2D list, NumPy array**
* Decision: NumPy array
* Pros: Efficient manipulation, simple syntax.
* Cons: External library needed.
* **Game Logic:**
* Options: Procedural, Object-oriented
* Decision: Object-oriented
* Pros: Better organization, easier to extend.
* Cons: More complex initially.
* **User Interaction:**
* Options: Buttons, Drag-and-drop
* Decision: Buttons
* Pros: Simple, intuitive, easy to implement.
* Cons: Less dynamic.
* **Win Detection:**
* Options: Iterative checks, Recursive checks
* Decision: Iterative checks
* Pros: Simple, efficient for fixed-size board.
* Cons: Less flexible for different sizes.
* How did the extension affect your design?
* **Menu Options (New Game, Restart Game, Exit):**
* Impact: Required additional GUI elements and methods for game control.
* Changes: Added create\_menu method and menu setup in the constructor.
* **Turn Indicator:**
* Impact: Needed a dynamic label to display the current player's turn.
* Changes: Added info\_label to the GUI and update\_info\_label method to update the text.
* **Win Announcement:**
* Impact: Required a mechanism to detect game end and inform the user.
* Changes: Modified handle\_move to check for win condition and show a message box.
* **Game Reset (New Game/Restart):**
* Impact: Required functionality to reset the game state and UI.
* Changes: Added new\_game and restart\_game methods to reset the board and update UI components.
* Describe your tests (e.g., what you tested, equivalence classes).
* **Test: test\_create\_board**
* What it tests: Ensures the board is initialized correctly.
* Equivalence Class: Empty board initialization.
* Method: Verifies that the board is a 6x7 matrix filled with zeros.
* **Test: test\_drop\_piece**
* What it tests: Validates that a piece is correctly placed on the board.
* Equivalence Class: Piece dropping in an empty column.
* Method: Drops a piece in the first column and checks if it appears in the correct position.
* **Test: test\_is\_valid\_location**
* What it tests: Checks if a column can accept a new piece.
* Equivalence Classes:
* Empty column.
* Full column.
* Method:
* Initially confirms the column is valid.
* Fills the column and then confirms it is no longer valid.
* **Test: test\_get\_next\_open\_row**
* What it tests: Finds the next available row in a column.
* Equivalence Classes:
* Empty column.
* Partially filled column.
* Method:
* Checks the next open row for an empty column.
* Drops a piece and then checks the next open row again.
* **Test: test\_winning\_move**
* What it tests: Detects a winning move on the board.
* Equivalence Classes:
* Horizontal win condition.
* Method:
* Drops pieces in a row to create a horizontal win condition and verifies if the winning move is detected.
* What lessons did you learn from the comprehensive exercise (i.e., programming concepts, software process)?
* **Modularity:**
* First, we learnt to break the game into **functions and classes** improves organization and maintainability.
* **Testing:**
* Writing unit tests is crucial for verifying functionality and catching errors early.
* **User Interface Design:**
* Simplifying UI elements (like using buttons) enhances user experience and ease of implementation.
* **Object-Oriented Programming:**
* Encapsulation and class-based design help manage complex logic and state changes efficiently.
* Iterative Development:
* Building the project incrementally allows for continuous testing and improvement.
* **Library Utilization:**
* Leveraging libraries like Tkinter and NumPy simplifies tasks and boosts efficiency.
* **Problem-Solving:**
* Addressing specific challenges, such as win detection, enhances critical thinking and algorithm design skills.
* **Code Readability:**
* Writing clear and concise code aids in collaboration and future maintenance.
* What functionalities are you going to demo?
* **Some** of the functionalities we are going to present:
* Creating a new game
* Dropping a piece into a column
* Winning the game
* Restarting the game
* Exiting the application
* Who is going to speak about each portion of your presentation? (Recall: Each group will have ten minutes to present their work; minimum length of group presentation is seven minutes. Each student must present for at least two minutes of the presentation.)
* Description
* Requirement analysis
* Design
* Implementation
* Testing
* Demonstration
* Lessons learnt.
* Other notes:
* <<Insert notes>>

<<Use your notes from above to complete create your slides and plan your presentation and demo.>>