MAPÚA UNIVERSITY
SCHOOL OF ELECTRICAL, ELECTRONICS, AND COMPUTER ENGINEERING

# Experiment 4:
# Design Patterns and Unit Testing

CPE106L (Software Design Laboratory)

**Brian Matthew E. Clemente**

**Maria Angelica Acantilado**

**Chalzea Fransen C. Dytianquin**

**Vance David G. Samia**

Group No.: **4**
Section: **FOPI01**

## PreLab

### Readings, Insights, and Reflection

**METIS #1:**
**Laura Cassell & Alan Gauld. *Python Projects (Edition 1)*. Wiley Professional Development (P&T). ISBN: 9781118909195.**

**Chapter 4: Building Desktop Applications, MVC Pattern - Page 221**
The group recognizes the critical role of Chapter 4 in *Python Projects* by Laura Cassell and Alan Gauld in elucidating the Model-View-Controller (MVC) pattern as a fundamental architectural framework for desktop application development. The chapter underscores how the deliberate separation of concerns within an application enhances modularity, maintainability, and scalability, thereby fostering efficient software engineering practices. By compartmentalizing functionality into the model, view, and controller layers, the MVC pattern streamlines debugging, facilitates testing, and optimizes collaborative development. The practical examples presented in the text offer a comprehensive perspective on its implementation, reinforcing its applicability in real-world software design. While the intricacies of delineating responsibilities within the MVC framework may pose initial challenges, the group acknowledges its indispensable role in constructing well-structured and adaptable applications. The insights gained from this chapter serve as a valuable foundation for refining software development methodologies and adhering to best practices in system architecture.

==========================================================

**METIS #2:**
**Sale, D. (2014). *Testing Python*. Wiley Professional, Reference & Trade (Wiley K&L). https://bookshelf.vitalsource.com/books/9781118901243**

The group recognizes the pivotal role of software testing in ensuring the reliability, efficiency, and maintainability of Python applications, as emphasized in *Testing Python* by David Sale. The text provides a comprehensive exploration of various testing methodologies, including unit testing, integration testing, and test-driven development (TDD), which are essential for identifying defects and enhancing code quality. By systematically examining testing frameworks and best practices, the book underscores the importance of automation in streamlining the debugging process and mitigating potential system failures. The structured approach to testing outlined in the text reinforces its significance in professional software development, where rigorous validation procedures are crucial for sustaining high-performance applications. Although the complexity of implementing advanced testing techniques may pose challenges, the group acknowledges that mastering these principles is fundamental to producing robust and scalable software solutions.

================================================================

**METIS #3:**
**Luke Sneeringer. *Professional Python (Edition 1)*. Wiley Professional Development (P&T). ISBN: 9781119070832**

**Unit Testing, (Chap 11)**
The chapter provides a structured examination of unit testing as a fundamental practice in software development, emphasizing its role in ensuring code reliability, maintainability, and early defect detection. By systematically breaking down the principles of test case design, assertion mechanisms, and testing frameworks, the text highlights best practices for implementing effective unit tests. Additionally, the discussion on automation and continuous integration underscores the necessity of integrating unit testing into the software development lifecycle. While the intricacies of writing comprehensive test cases may present challenges, the group recognizes that mastering unit testing methodologies is essential for developing robust and scalable Python applications. The insights from this chapter reinforce the necessity of adopting disciplined testing practices to enhance software quality and prevent unforeseen system failures.

================================================================

## METIS Book #1

**Clemente (Chapter 4)**
Chapter 4 of *Python Projects* by Laura Cassell and Alan Gauld provides a structured approach to building desktop applications, emphasizing the importance of separating core logic, data processing, and presentation layers. The discussion of the Model-View-Controller (MVC) pattern highlights the advantages of modularity, allowing developers to create multiple user interfaces without altering the core functionality. The transition from a console-based Tic-Tac-Toe application to a graphical interface using Tkinter illustrates how GUI development enhances user experience while maintaining the same logical foundation. The chapter also introduces alternative GUI frameworks, reinforcing the idea that choosing the right toolkit can significantly impact an application's efficiency and usability.

Beyond UI development, the material explores essential software engineering concepts, such as handling configuration data, implementing logging mechanisms, and localizing applications for a broader audience, The inclusion of Unicode support and translation tools like `gettext` underscores the importance of accessibility in modern software. Overall, this chapter provides valuable insights into structuring applications effectively, demonstrating how well-organized code can lead to scalable and maintainable software. The hands-on projects reinforce these concepts, making the learning process more practical and applicable.

**Acantilado (Chapter 4)**
Chapter 4 provides essential guidance on structuring and developing desktop applications using the Python programming language. It emphasizes the importance of organizing an application into three fundamental components: the user interface, the core logic, and the data layer; it ensures that applications are scalable, maintainable, and efficient in the long term. A key principle highlighted in the chapter is the recommendation to initially develop applications using a command-line interface before transitioning to a graphical user interface with the Tkinter library. This method allows developers to concentrate on refining the core functionality of the application before focusing on enhancing usability through visual elements. By following this approach, programmers can create more robust and well-structured software solutions. Additionally, the chapter explores the flexibility of Python in desktop application development, emphasizing the availability of third-party frameworks that provide advanced tools for building sophisticated and high-performing applications. These external frameworks extend Python’s capabilities, enabling developers to create feature-rich software while maintaining efficiency and scalability.

**Dytianquin (Chapter 4)**
Chapter 4 of *Python Projects* by Laura Cassell and Alan Gauld presents a systematic approach to developing desktop applications, emphasizing the necessity of separating an application's core logic, data processing, and user interface components. The discussion of the Model-View-Controller architectural pattern highlights the benefits of modularity, enabling developers to design multiple user interfaces without modifying the underlying logic. An illustrative example in the chapter is the transformation of a text-based Tic-Tac-Toe game into a graphical interface using the Tkinter library, demonstrating how graphical user interface development enhances user experience while preserving the same fundamental functionality. The chapter also introduces alternative graphical frameworks, reinforcing the significance of selecting an appropriate toolkit to optimize both application performance and usability. In addition to user interface development, the chapter delves into essential software engineering principles, including configuration management, logging implementation, and localization strategies to accommodate a global user base. The inclusion of Unicode compatibility and translation utilities such as the gettext module underscores the importance of accessibility and internationalization in contemporary software development.

**Samia (Chapter 4)**
Laura Cassell and Alan Gauld's Python Projects, Chapter 4, offers helpful advice on organizing and creating desktop Python apps. It highlights how crucial it is to divide an application into its user interface, core logic, and data layer in order to produce scalable and maintainable systems. The ability to begin with a command-line interface (CLI) and then switch to a graphical user interface (GUI) using Tkinter is a crucial lesson learned from this chapter. With this method, developers can concentrate on the logic of the application first and then use a graphical user interface to improve usability. Furthermore, Python's versatility and the range of tools available for developing sophisticated desktop applications are highlighted by the development of third-

party frameworks. By providing structured guidance and hands-on projects, the chapter offers practical insights into building well-organized, scalable, and maintainable applications, ensuring that developers acquire both theoretical knowledge and practical experience in designing robust software systems.

## METIS Book #2

**Clemente (Pages 1 – 12)**

*Testing Python* by David Sale explores the evolution of software testing and its growing importance in modern development. The book highlights how testing has shifted from being an afterthought to an essential practice, driven by the increasing complexity of software and advancements in technology. This shift has led to more structured testing methodologies that prioritize code stability and quality. One key takeaway is the importance of automating tests to catch errors early and streamline the development process.

Likewise, the book also introduces popular testing frameworks and best practices, such as Behavior-Driven Development (BDD) and Test-Driven Development (TDD), which reinforce the idea that writing testable code is just as important as writing functional code. Overall, *Testing Python* reinforces the necessity of adopting a testing-first mindset. While thorough testing may initially seem time-consuming, it ultimately leads to more secure, maintainable, and efficient software – an essential requirement in today's rapidly evolving digital landscape.

**Acantilado (Pages 1 – 12)**

David Sale's *Testing Python* delves into the evolution of software testing, emphasizing its transformation from an optional step to an integral part of the development lifecycle. The book discusses how the increasing complexity of modern software, along with advancements in technology, has necessitated more structured approaches to ensure code reliability and stability. A major focus is placed on the role of automation in testing, as early detection of errors not only improves efficiency but also minimizes costly debugging later in the process. Furthermore, the book introduces key testing methodologies such as test-driven and behavior-driven development, stressing that writing code with testing in mind is just as crucial as its functionality. Ultimately, the book illustrates how testing frameworks early in the development process reduces security vulnerabilities and prevents code degradation.

**Dytianquin (Pages 1 – 12)**

The evolution of software testing marks a significant shift in how developers prioritize code quality, moving from reactive bug-fixing to a proactive, structured approach. Historically, testing was often overlooked, with errors addressed only after they surfaced. However, as software systems grew more complex and technological advancements accelerated, testing became an integral part of development. This shift, driven by improved computing power and a recognition of testing's role in software security and maintainability, has led to the widespread adoption of methodologies such as behavior-driven development and test-driven development. These approaches align with agile practices by promoting continuous feedback and early issue detection, reducing costly revisions later in the process.

**Samia (Pages 1 – 12)**
A notable change in the way developers approach code quality and stability can be seen in the history and development of software testing. Testing was frequently neglected in the past, and problems were only fixed after they were discovered. However, as software complexity rose and technology improved, testing became a crucial step in the development process. This change is a result of both advancements in computer power and a shift in perspective, as developers now understand that thorough testing produces software that is more secure, reliable, and maintainable. A proactive approach to testing is emphasized by the emergence of approaches like Behavior-Driven Development (BDD) and Test-Driven Development (TDD), which guarantee that code is validated at every level of development.

The underscored approaches fit in nicely with contemporary agile processes, which emphasize constant feedback and quick iteration. Teams may find and address problems early on and avoid expensive and time-consuming solutions later by integrating testing into the development process. To sum up, *Testing Python's* insights highlight how important it is for modern software development to have a testing-first mentality. Thorough testing saves time and effort by lowering faults and enhancing overall program reliability, even though it may first seem time-consuming. To produce high-quality software that satisfies the expectations of an increasingly complicated digital world, developers must embrace testing as a core discipline; it ultimately highlights how incorporating thorough testing practices from the outset leads to software that is more secure, maintainable, and adaptable to future changes

## METIS Book #3

**Clemente (Chapter 11)**
Chapter 11 of *Professional Python* by Luke Sneeringer underscores the critical role of unit testing in modern software development. It presents unit testing as an integral practice that ensures individual components of a program function correctly while maintaining code reliability and efficiency. Rather than treating testing as an afterthought, Sneeringer advocates for incorporating it into the development process from the outset. A key takeaway from this chapter is that unit tests provide developers with confidence in their code. Writing and running tests early in the development cycle helps identify issues before they escalate, leading to improved debugging and overall software quality.

**Acantilado (Chapter 11)**
Luke Sneeringer's *Professional Python* highlights unit testing as a crucial practice in modern software development, ensuring that individual components of a program function correctly.

Rather than being an afterthought, testing is presented as a core discipline that contributes to writing reliable, maintainable, and efficient code. One of the key takeaways from Sneeringer's discussion is that unit tests provide developers with confidence in their code by enabling early detection of errors and continuous validation throughout the development process. This proactive approach minimizes potential defects, enhances code quality, and simplifies debugging. Additionally, Sneeringer emphasizes the benefits of automated unit tests, which allow developers to modify and improve their code without the risk of unintentionally breaking existing functionality.

**Dytianquin (Chapter 11)**
Chapter 11 of *Professional Python* by Luke Sneeringer explores the significance of unit testing in contemporary software development, emphasizing its role in verifying the functionality of individual components while preserving code integrity and performance. Instead of considering testing as an optional phase, Sneeringer advocates for embedding it into the development workflow from the start to ensure software reliability. One of the key insights from this chapter is that unit testing instills confidence in developers by catching issues early, simplifying the debugging process, and ultimately improving code quality. Additionally, automated unit tests provide a safeguard against unintended disruptions when modifying code, reinforcing the necessity of a proactive testing strategy.

**Samia (Chapter 11)**
A key technique in contemporary software development is unit testing, which makes sure that each program's separate parts work as intended. Unit testing is a fundamental discipline that helps write dependable, maintainable, and effective code, according to Luke Sneeringer's book Professional Python. His explanation clarifies why testing ought to be considered an essential component of the development process rather than an afterthought.

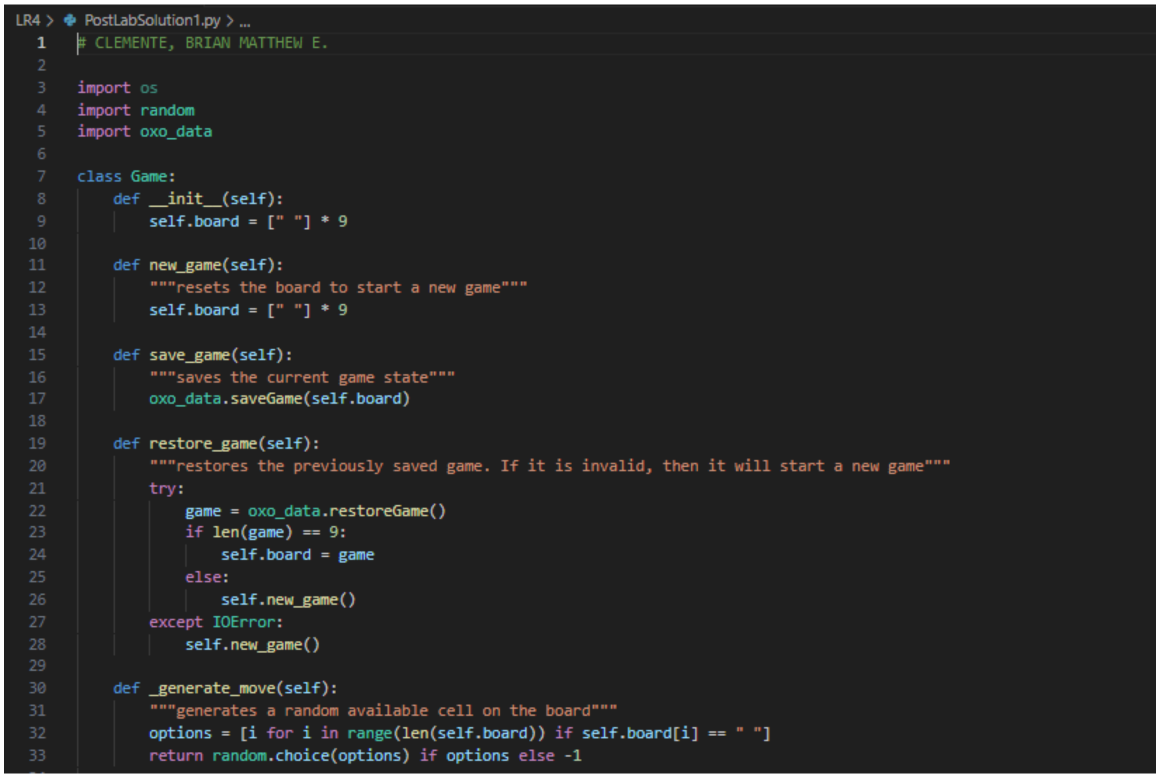
Unit tests provide developers confidence in their code, which is one important lesson to be learned from this chapter. Programmers can identify and address problems before they become more serious by creating tests early on and running them frequently throughout the development process. This proactive strategy reduces the range of possible errors, which not only enhances code quality but also facilitates debugging. Sneeringer also emphasizes how developers can change code without worrying about inadvertently disrupting existing functionality thanks to automated unit tests.

# PostLab

## Programming Problems

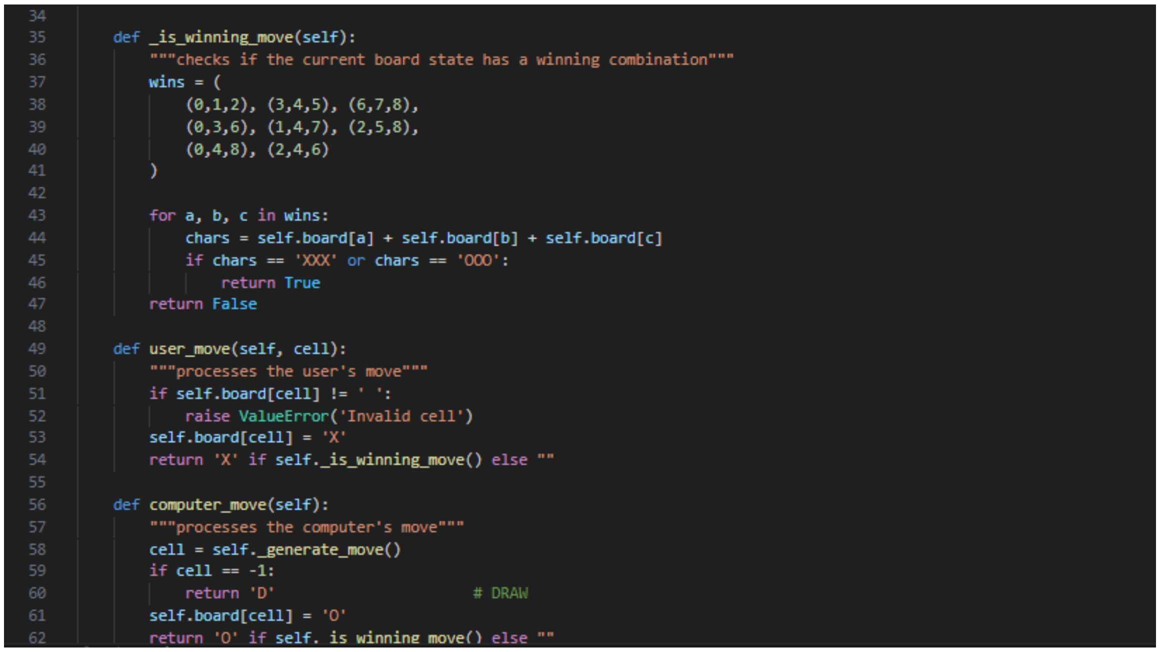
This post-lab activity focuses on the implementation of design patterns and unit testing, reinforcing key principles in software architecture, modularity, and code reliability. The tasks involve applying commonly used design patterns, such as Model-View-Controller (MVC), to create scalable and maintainable software solutions. Additionally, unit testing is integrated to validate the correctness of individual components, ensuring that modifications do not introduce unintended errors. Each programming task was distributed among group members, with contributions documented within the code files to maintain clarity and organization. The development process emphasized structured modeling techniques, enabling a systematic approach to implementing reusable and efficient design patterns. Version control tools were used to track changes, ensuring seamless collaboration and code integrity.

### Programming Problem #1



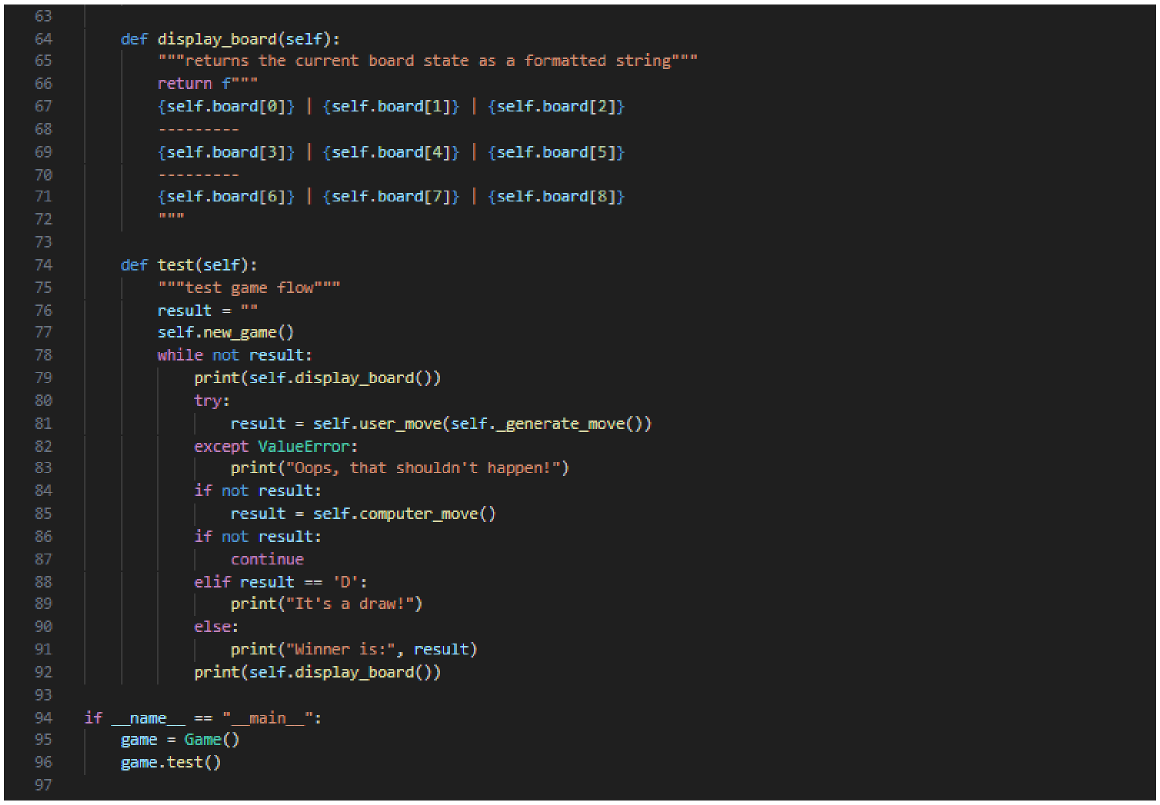
```python
# CLEMENTE, BRIAN MATTHEW E.

import os
import random
import oxo_data

class Game:
    def __init__(self):
        self.board = [" "] * 9

    def new_game(self):
        """resets the board to start a new game"""
        self.board = [" "] * 9

    def save_game(self):
        """saves the current game state"""
        oxo_data.saveGame(self.board)

    def restore_game(self):
        """restores the previously saved game. If it is invalid, then it will start a new game"""
        try:
            game = oxo_data.restoreGame()
            if len(game) == 9:
                self.board = game
            else:
                self.new_game()
        except IOError:
            self.new_game()

    def _generate_move(self):
        """generates a random available cell on the board"""
        options = [i for i in range(len(self.board)) if self.board[i] == " "]
        return random.choice(options) if options else -1
```

(a)



```
    def _is_winning_move(self):
        """checks if the current board state has a winning combination"""
        wins = (
            (0,1,2), (3,4,5), (6,7,8),
            (0,3,6), (1,4,7), (2,5,8),
            (0,4,8), (2,4,6)
        )

        for a, b, c in wins:
            chars = self.board[a] + self.board[b] + self.board[c]
            if chars == 'XXX' or chars == 'OOO':
                return True
        return False

    def user_move(self, cell):
        """processes the user's move"""
        if self.board[cell] != ' ':
            raise ValueError('Invalid cell')
        self.board[cell] = 'X'
        return 'X' if self._is_winning_move() else ""

    def computer_move(self):
        """processes the computer's move"""
        cell = self._generate_move()
        if cell == -1:
            return 'D'                      # DRAW
        self.board[cell] = 'O'
        return 'O' if self._is_winning_move() else ""
```

(b)

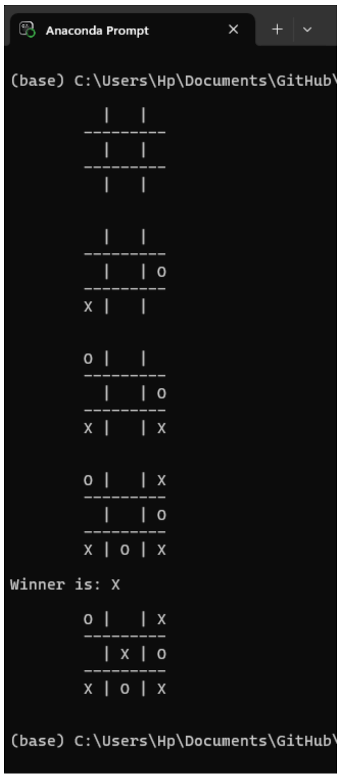


```

    def display_board(self):
        """returns the current board state as a formatted string"""
        return f"""
        {self.board[0]} | {self.board[1]} | {self.board[2]}
        ---------
        {self.board[3]} | {self.board[4]} | {self.board[5]}
        ---------
        {self.board[6]} | {self.board[7]} | {self.board[8]}
        """

    def test(self):
        """test game flow"""
        result = ""
        self.new_game()
        while not result:
            print(self.display_board())
            try:
                result = self.user_move(self._generate_move())
            except ValueError:
                print("Oops, that shouldn't happen!")
            if not result:
                result = self.computer_move()
            if not result:
                continue
            elif result == 'D':
                print("It's a draw!")
            else:
                print("Winner is:", result)
            print(self.display_board())

if __name__ == "__main__":
    game = Game()
    game.test()

```

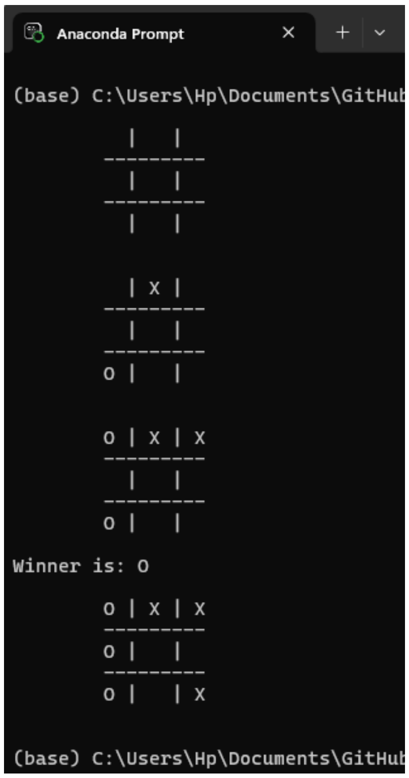
(c)

*Figure 1.1 Screenshots of the Programming Problem #1 Source Code Based on the oxo_logic.py Reference File*

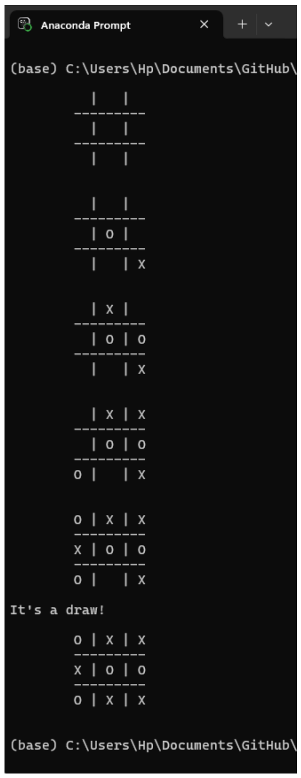
*Figure 1.1* displays the source code for the Tic-Tac-Toe game, showcasing the core logic that governs gameplay. The code follows an object-oriented programming (OOP) approach, encapsulating game functionality within a `Game` class, building upon the provided reference file, `oxo_logic.py`. Key methods include `user_move()` and `computer_move()`, which handle player and AI turns, respectively, while _is_winning_move() checks for a winning combination after each move. Additionally, `save_game()` and `restore_game()` allow for saving and loading game progress. The structure of the code ensures modularity, reusability, and maintainability, making it easier to extend or modify. The screenshots effectively illustrate the program's logic, highlighting how user input, computer-generated moves, and game state management work together to determine the game's outcome.



```
(base) C:\Users\Hp\Documents\GitHub\
          |   |
        ---------
          |   |
        ---------
          |   |

          |   |
        ---------
          |   | O
        ---------
        X |   |

        O |   |
        ---------
          |   | O
        ---------
        X |   | X

        O |   | X
        ---------
          |   | O
        ---------
        X | O | X
Winner is: X
        O |   | X
        ---------
          | X | O
        ---------
        X | O | X

(base) C:\Users\Hp\Documents\GitHub\
```



```
(base) C:\Users\Hp\Documents\GitHub
          |   |
        ---------
          |   |
        ---------
          |   |

          | X |
        ---------
          |   |
        ---------
        O |   |

        O | X | X
        ---------
          |   |
        ---------
        O |   |
Winner is: O
        O | X | X
        ---------
        O |   |
        ---------
        O |   | X

(base) C:\Users\Hp\Documents\GitHub
```



```
(base) C:\Users\Hp\Documents\GitHub\
          |   |
        ---------
          |   |
        ---------
          |   |

          |   |
        ---------
          | O |
        ---------
          |   | X

          | X |
        ---------
          | O | O
        ---------
          |   | X

          | X | X
        ---------
          | O | O
        ---------
        O |   | X

        O | X | X
        ---------
        X | O | O
        ---------
        O |   | X
It's a draw!
        O | X | X
        ---------
        X | O | O
        ---------
        O | X | X

(base) C:\Users\Hp\Documents\GitHub\
```

(a) (b) (c)

*Figure 1.2 Screenshots of the Programming Problem #1 Solution Outputs Executed in Anaconda Prompt*

*Figure 1.2* illustrates the three possible outcomes: (a) X wins, (b) O wins, and (c) Draw. Each outcome is determined by the sequences of moves made by the player (X) and the computer (O).

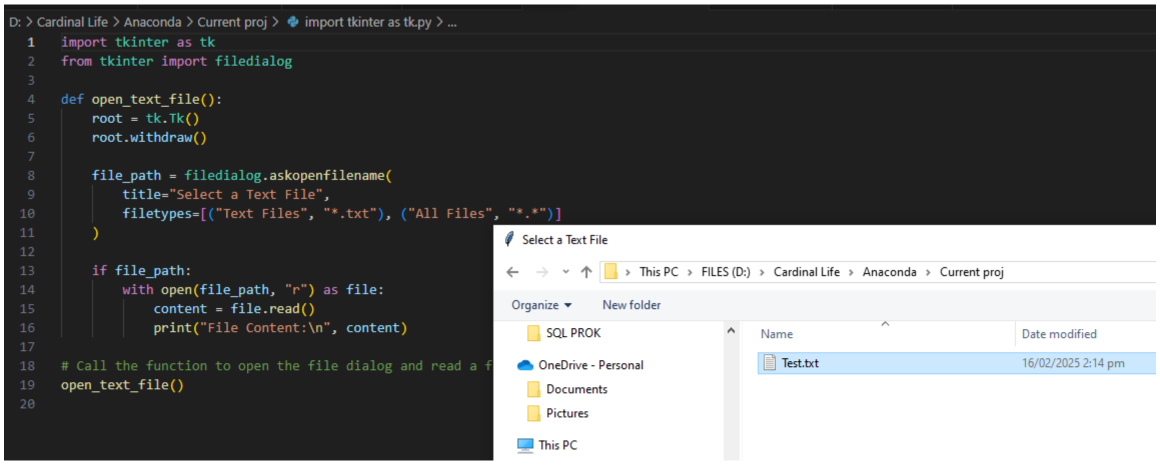
In screenshot (a), X wins, the player successfully aligns three `X` marks in a winning pattern. When this condition is met, the `_is_winning_move()` function detects the victory, and the program outputs "`Winner is: X`". This outcome represents a strategic success for the player, who managed to place their marks in a winning formation before the computer could block the move.

In screenshot (b), O wins, the computer achieves a winning combination before the player, leading to the output “`Winner is: O`”. Similar to the previous case, the `_is_winning_move()` function verifies the win, and the computer secures victory by placing three `O` marks in a winning pattern. Since the computer's moves are randomly generated, this outcome demonstrates the element of unpredictability in the game.

Lastly, screenshot (c) represents a Draw, where all board cells are occupied without either player forming a winning sequence. The game recognizes this situation when `_generate_move()` returns `-1`, indicating that no more moves are possible. In response, the program outputs `"It's a draw"`. A draw suggests that both players played optimally, effectively blocking each other from winning.

These three outcomes – winning for X, winning for O, or a draw – highlight the dynamic nature of the game. While a player's success depends on their strategy, the computer's randomness adds a layer of unpredictability, making each match a unique experience.

## Programming Problem #2


D: > Cardinal Life > Anaconda > Current proj > import tkinter as tk.py > ...

```python
import tkinter as tk
from tkinter import filedialog

def open_text_file():
    root = tk.Tk()
    root.withdraw()

    file_path = filedialog.askopenfilename(
        title="Select a Text File",
        filetypes=[("Text Files", "*.txt"), ("All Files", "*.*")]
    )

    if file_path:
        with open(file_path, "r") as file:
            content = file.read()
            print("File Content:\n", content)

# Call the function to open the file dialog and read a f
open_text_file()
```

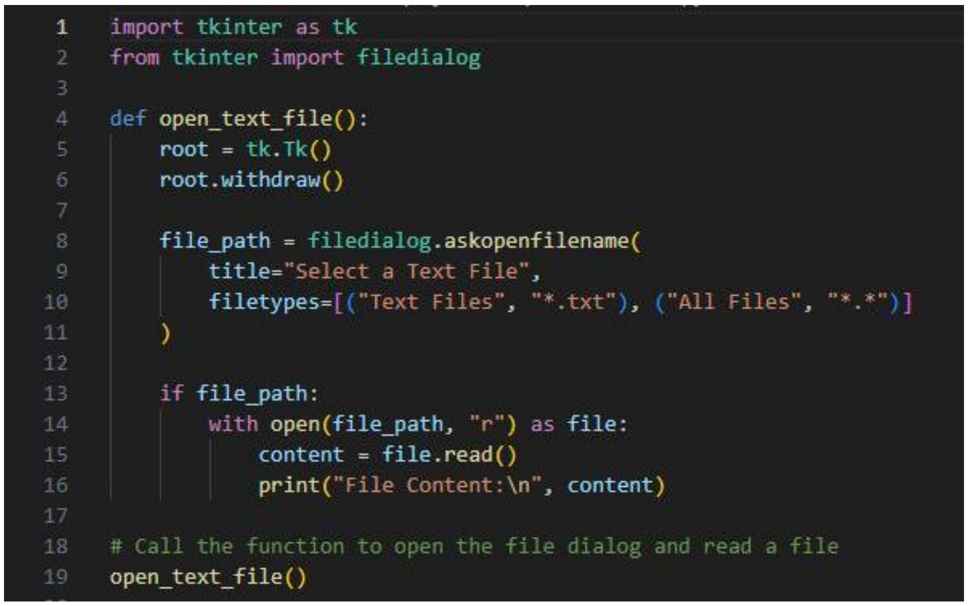
Select a Text File

This PC > FILES (D:) > Cardinal Life > Anaconda > Current proj

| Name | Date modified |
|---|---|
| Test.txt | 16/02/2025 2:14 pm |

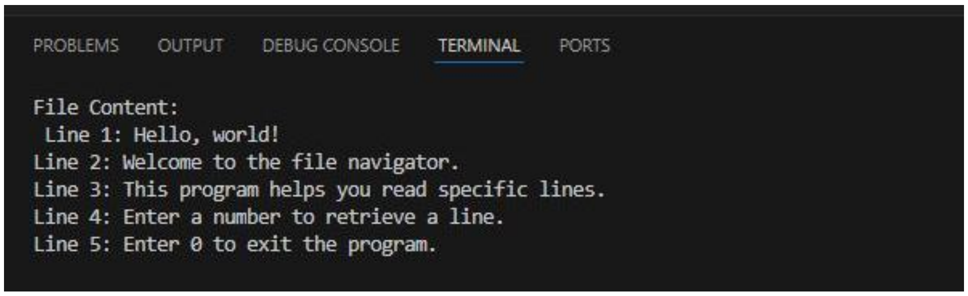
*Figure 2.1 Screenshot of Tkinter prompt for text file selection.*

*Figure 2.1* displays the Tkinter file dialog that appears when the application is executed, providing users with a familiar and intuitive file selection interface. Through a recognizable file explorer window, users can navigate their system, browse directories, and select a text file of their choice. The `tkinter.filedialog.askopenfilename()` function is responsible for opening this dialog, allowing users to conveniently choose a file without manually entering its path. Once a file has been selected, the application can proceed to process its contents, including reading, analyzing, and displaying the text. This functionality is particularly useful for a variety of applications, such as data processing tools, text editors, or file management programs, where seamless file selection and retrieval enhance overall user experience and efficiency.



```python
import tkinter as tk
from tkinter import filedialog

def open_text_file():
    root = tk.Tk()
    root.withdraw()

    file_path = filedialog.askopenfilename(
        title="Select a Text File",
        filetypes=[("Text Files", "*.txt"), ("All Files", "*.*")]
    )

    if file_path:
        with open(file_path, "r") as file:
            content = file.read()
            print("File Content:\n", content)

# Call the function to open the file dialog and read a file
open_text_file()
```

(a)



```
File Content:
 Line 1: Hello, world!
Line 2: Welcome to the file navigator.
Line 3: This program helps you read specific lines.
Line 4: Enter a number to retrieve a line.
Line 5: Enter 0 to exit the program.
```

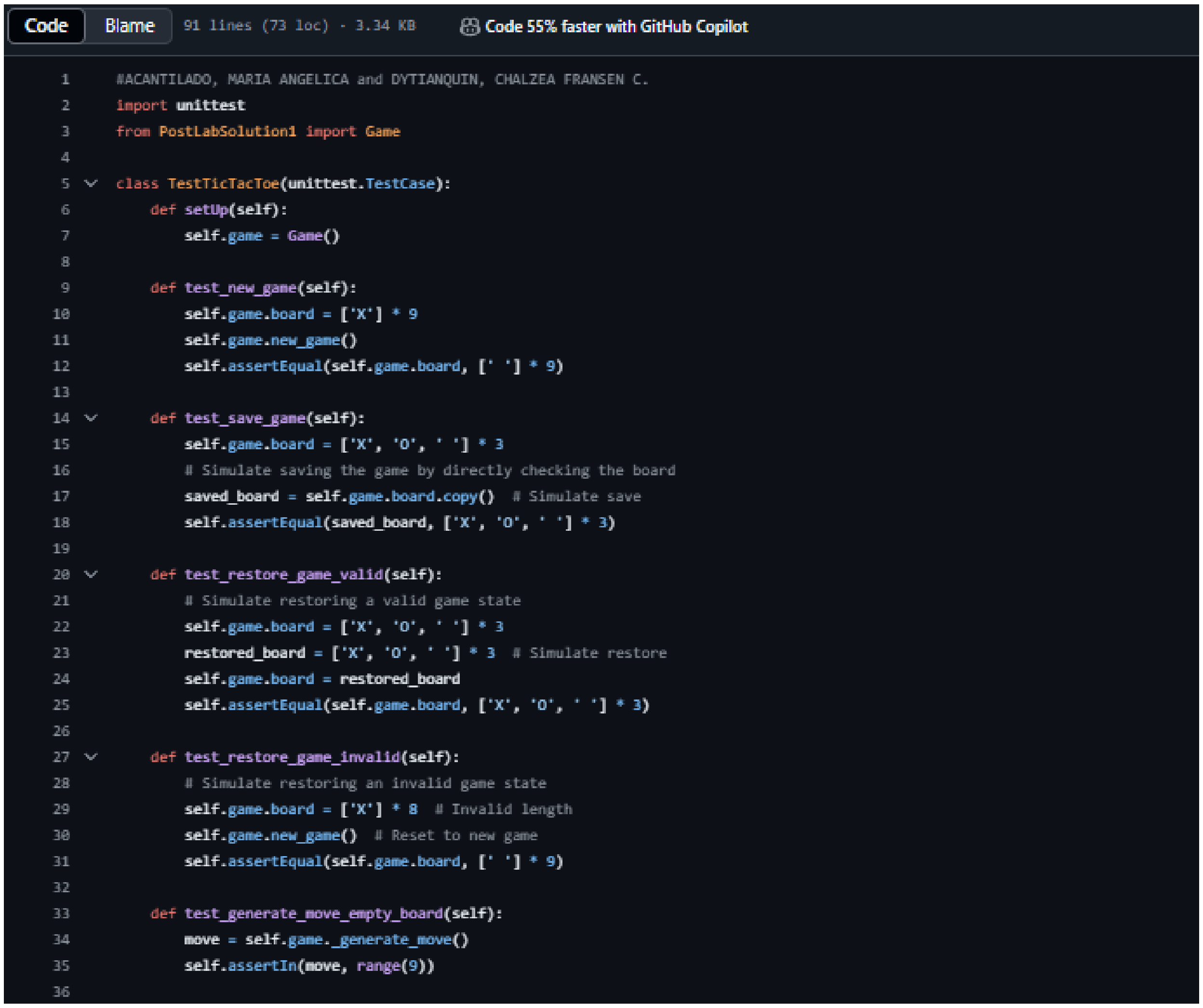
(b)

*Figure 2.2 Screenshot of the Programming Problem #2 Source Code and its Output of Selected Text File's Contents*

*Figure 2.2* presents a screenshot of the Python source code responsible for implementing a file selection process using Tkinter. The code snippet shown in part (a) demonstrates how the `tkinter.filedialog` module is used to prompt the user to select a text file from their system. Once a file is chosen, the program employs Python's built-in `open()` function to read its contents. The read data is then stored in a variable and subsequently printed to the console. This implementation streamlines the process of accessing text files by providing a user-friendly graphical interface for selection, eliminating the need for manual file path entry. Likewise, in part (b) of the same figure, the corresponding output of the program is displayed within a terminal window, showing the extracted text from the selected file.

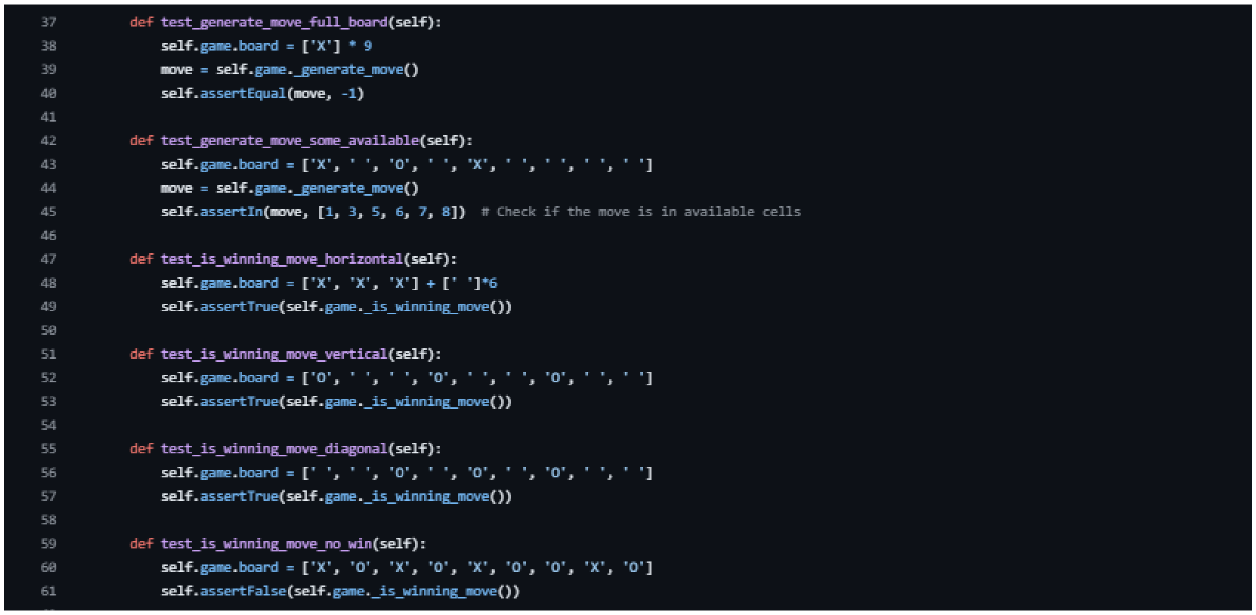
The output presents the file's content in a structured format, where each line is labeled numerically, enhancing readability and usability. This feature allows users to review the file's content efficiently and serves as a foundation for further text-processing functionalities, such as searching for specific lines or modifying the text. The combination of Tkinter for file selection and Python's file-handling capabilities results in an effective method for interacting with text files through both graphical and command-line interfaces.

## Programming Problem #3


Code
Blame
91 lines (73 loc) · 3.34 KB
Code 55% faster with GitHub Copilot

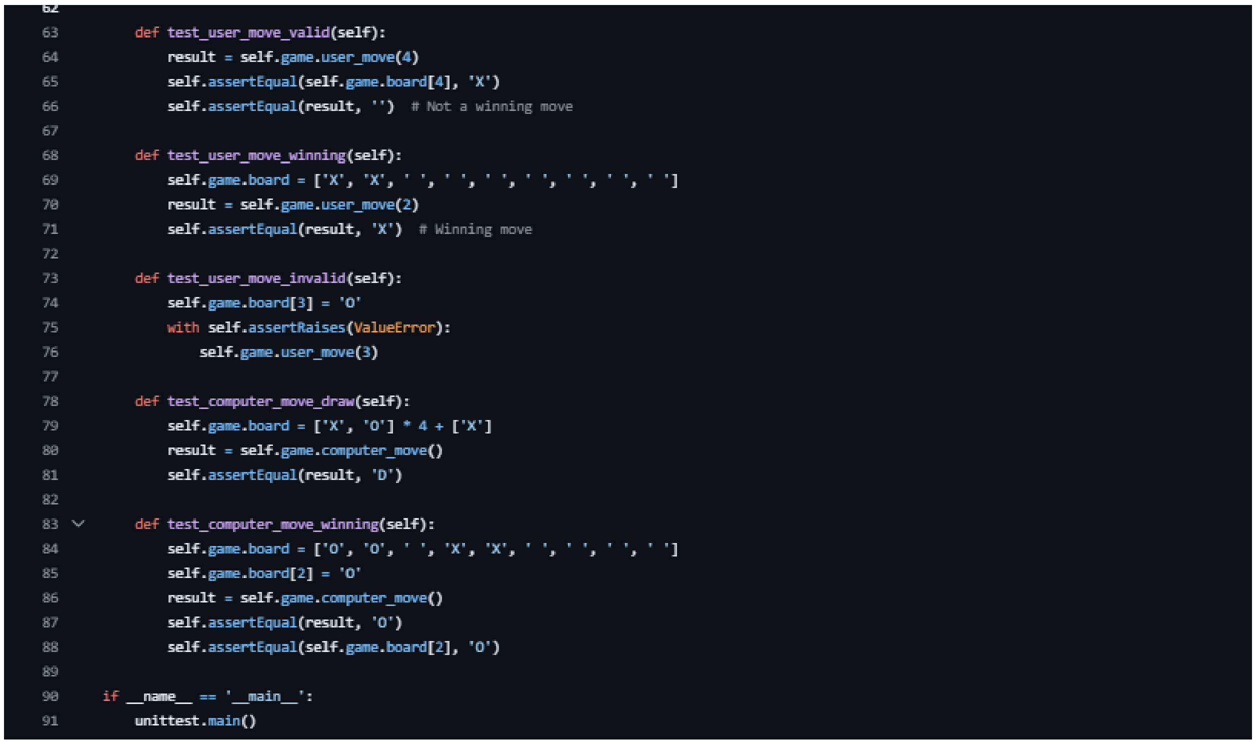
```python
#ACANTILADO, MARIA ANGELICA and DYTIANQUIN, CHALZEA FRANSEN C.
import unittest
from PostLabSolution1 import Game

class TestTicTacToe(unittest.TestCase):
    def setUp(self):
        self.game = Game()

    def test_new_game(self):
        self.game.board = ['X'] * 9
        self.game.new_game()
        self.assertEqual(self.game.board, [' '] * 9)

    def test_save_game(self):
        self.game.board = ['X', 'O', ' '] * 3
        # Simulate saving the game by directly checking the board
        saved_board = self.game.board.copy()  # Simulate save
        self.assertEqual(saved_board, ['X', 'O', ' '] * 3)

    def test_restore_game_valid(self):
        # Simulate restoring a valid game state
        self.game.board = ['X', 'O', ' '] * 3
        restored_board = ['X', 'O', ' '] * 3  # Simulate restore
        self.game.board = restored_board
        self.assertEqual(self.game.board, ['X', 'O', ' '] * 3)

    def test_restore_game_invalid(self):
        # Simulate restoring an invalid game state
        self.game.board = ['X'] * 8  # Invalid length
        self.game.new_game()  # Reset to new game
        self.assertEqual(self.game.board, [' '] * 9)

    def test_generate_move_empty_board(self):
        move = self.game._generate_move()
        self.assertIn(move, range(9))

```

(a)



```python
    def test_generate_move_full_board(self):
        self.game.board = ['X'] * 9
        move = self.game._generate_move()
        self.assertEqual(move, -1)

    def test_generate_move_some_available(self):
        self.game.board = ['X', ' ', 'O', ' ', 'X', ' ', ' ', ' ', ' ']
        move = self.game._generate_move()
        self.assertIn(move, [1, 3, 5, 6, 7, 8])  # Check if the move is in available cells

    def test_is_winning_move_horizontal(self):
        self.game.board = ['X', 'X', 'X'] + [' ']*6
        self.assertTrue(self.game._is_winning_move())

    def test_is_winning_move_vertical(self):
        self.game.board = ['O', ' ', ' ', 'O', ' ', ' ', 'O', ' ', ' ']
        self.assertTrue(self.game._is_winning_move())

    def test_is_winning_move_diagonal(self):
        self.game.board = [' ', ' ', 'O', ' ', 'O', ' ', 'O', ' ', ' ']
        self.assertTrue(self.game._is_winning_move())

    def test_is_winning_move_no_win(self):
        self.game.board = ['X', 'O', 'X', 'O', 'X', 'O', 'O', 'X', 'O']
        self.assertFalse(self.game._is_winning_move())
```

(b)

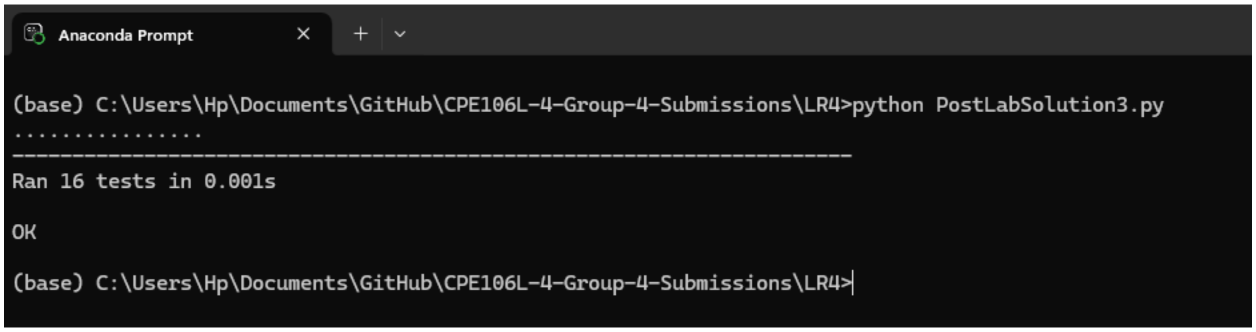


```python
    def test_user_move_valid(self):
        result = self.game.user_move(4)
        self.assertEqual(self.game.board[4], 'X')
        self.assertEqual(result, '')  # Not a winning move

    def test_user_move_winning(self):
        self.game.board = ['X', 'X', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
        result = self.game.user_move(2)
        self.assertEqual(result, 'X')  # Winning move

    def test_user_move_invalid(self):
        self.game.board[3] = 'O'
        with self.assertRaises(ValueError):
            self.game.user_move(3)

    def test_computer_move_draw(self):
        self.game.board = ['X', 'O'] * 4 + ['X']
        result = self.game.computer_move()
        self.assertEqual(result, 'D')

    def test_computer_move_winning(self):
        self.game.board = ['O', 'O', ' ', 'X', 'X', ' ', ' ', ' ', ' ']
        self.game.board[2] = 'O'
        result = self.game.computer_move()
        self.assertEqual(result, 'O')
        self.assertEqual(self.game.board[2], 'O')

if __name__ == '__main__':
    unittest.main()
```

(c)

*Figure 3.1 Screenshots of the Programming Problem #3 Source Code for a unit test program for the Tic-Tac-Toe Console App*

Ultimately, the confines of *Figure 3.1* present the final source code for a unit test program designed to evaluate the functionality of the Tic-Tac-Toe Console Application. This unit testing framework systematically verifies critical aspects of the game, such as move validation, detection of winning conditions, and identification of draw scenarios. By implementing automated tests, developers can ensure the accuracy and reliability of the game's logic while minimizing the risk of undetected errors. Moreover, unit testing facilitates code maintainability by enabling developers to make modifications with confidence, as the tests help identify unintended changes that could compromise the program's functionality. This approach aligns with best practices in software development, emphasizing the importance of rigorous testing in producing robust and error-free applications.



```
(base) C:\Users\Hp\Documents\GitHub\CPE106L-4-Group-4-Submissions\LR4>python PostLabSolution3.py
................
----------------------------------------------------------------------
Ran 16 tests in 0.001s

OK

(base) C:\Users\Hp\Documents\GitHub\CPE106L-4-Group-4-Submissions\LR4>
```

*Figure 3.2 Screenshot of the Programming Problem #3 Output in Anaconda Prompt*

*Figure 3.2* displays the output of the unit test program for the Tic-Tac-Toe Console Application as executed in Anaconda prompt. The results indicate whether the game's core functionalities—such as valid move execution, win detection, and draw conditions—perform as expected. Each test case runs automatically, verifying the correctness of the implemented logic and highlighting any failures that may require debugging. This structured testing approach ensures that the program adheres to expected behavior, improving reliability and maintainability. By using command-line output, developers can efficiently review test results, making it easier to identify and address potential issues in the game's implementation.