

Assessment 02

COURSE CODE: S225 PRT582

SOFTWARE ENGINEERING: PROCESS AND TOOLS

**Software Unit Testing Report**

Location: Darwin Campus (ECP)

**Hangman Game Implementation using Test-Driven Development**

Student Name: Amrit Niure

Student ID: 396426

Course: PRT582 Software Engineering Process and Tools

Institution: Charles Darwin University

Assignment: A2 Individual Software Unit Testing Report (30%)

Date: September 5, 2025

GitHub Repository: <https://github.com/amrit-niure/hangman-with-tdd>

# Introduction

This report documents the implementation and development of Hangman Game with Test Driven Development approach and its principles using automated unit testing in Python. In this project I have demonstrated the practical application of Test Driven Development methodology where we write what should be the outcome of the code and then implement the actual process later which is demonstrated with writing the test code first outlining what should be the expected outcome and then implement the code to obtain that outcome.

# Programming Language and Tool

**Python** is used in this project to execute the hangman game and it has very simple and readable syntax structure and the development can be done at very fast pace and more code can be written in less with with maximum functionality .

For testing purposes I have used , **Pytest** and for linting the code for clean and maintainable code structure I have used **Flake8** and **pylint**.

In addition to this, for bundling the package of the game with various modules and managing the project dependencies , I have used **“uv”** which is a new package manager built on top of “**pip”,** which as industry standard for managing python project but “uv” is a lot more efficient and faster compared to “pip”.

I have also used **GitHub Actions** for **Continuous Integration (CI)** in order to integrate the latest changes with the existing code where I have set up a pipeline to run the test every time I push the changes to the github. It makes the communication and the workflow very smooth and maintainable.

In my Github Repository, there is a file *“.github/workflows/ci.yml”* that contains the project Continuous Integration pipeline.

# Process

The development of Hangman game followed strict Test Driven Development methodology, as evidenced by Git commit history and the screenshots I have provided in the github repository. Each feature was implemented using **Red-Green-Refactor** Lifecycle

**Red Phase:** Write a new failing test, that should represent the expected structure and logic for a feature that does not exists.

**Green Phase**: Write a minimum amount of code to make the failing test pass.

**Refactor Phase:** Improve and clean up the newly written code while making sure that all the tests passes.

**Two Levels Masking (Basic and Intermediate )**

The implementation should handle the two levels of the game which are basic as the single word guessing and intermediate as the phrase guessing. The masking feature should mask all the words and phrase. For instance if the phrase is “clean code!” , it should show “\_\_\_\_\_ \_\_\_\_!” .

**Red Phase**: The initial test failed because the masking logic treated all characters as letters to be hidden.

This was the first feature implemented. A test was written to ensure a simple word like "Python" would be correctly represented as six underscores.

**Red Phase**: This foundational test (*commit f6a4e1f*) failed because the Game class and its reveal attribute did not yet exist.

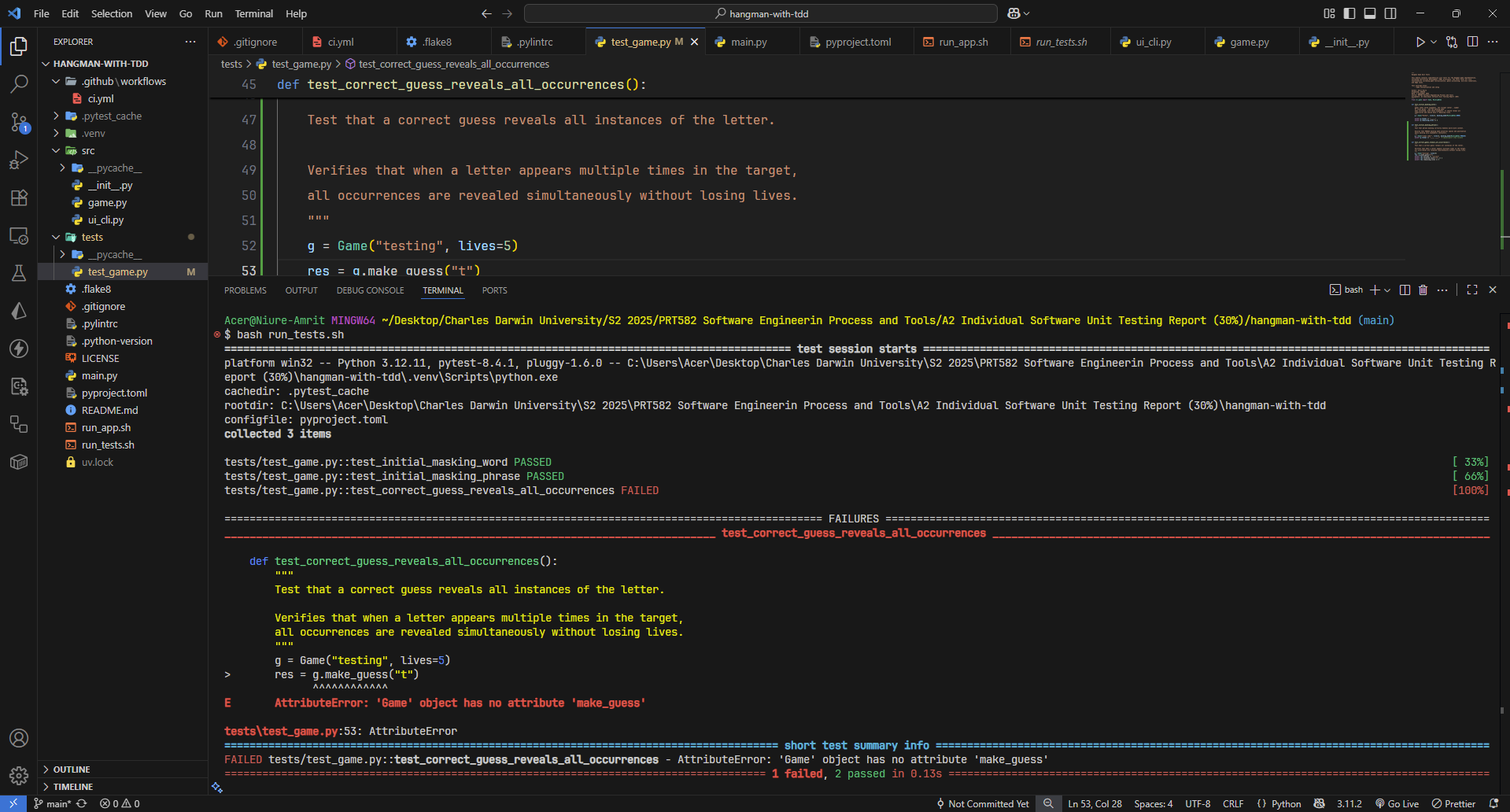
**Test Case**

def test\_initial\_masking\_word():

g = Game("Python", lives=3, masking\_mode=MaskingMode.WORD)

assert g.reveal == "\_\_\_\_\_\_"

assert g.remaining\_lives == 3



**Green Phase:** The Game class was created with a constructor and the \_mask() method to generate the initial underscore representation, allowing the test to pass.

**Test Case**

def test\_initial\_masking\_phrase():

g = Game("clean code!", lives=3, masking\_mode=MaskingMode.PHRASE)

assert g.reveal == "\_\_\_\_\_ \_\_\_\_!"

A screenshot of a computer program

AI-generated content may be incorrect.

**Green Phase:** To check the characteristics of the input character The \_mask() function was implemented. If the character is alphabetical , it is then replaced with an underscore and if it is not a character then it is skipped.

This is the implemented function :

def \_mask(self, text: str) -> str:

masked = []

for ch in text:

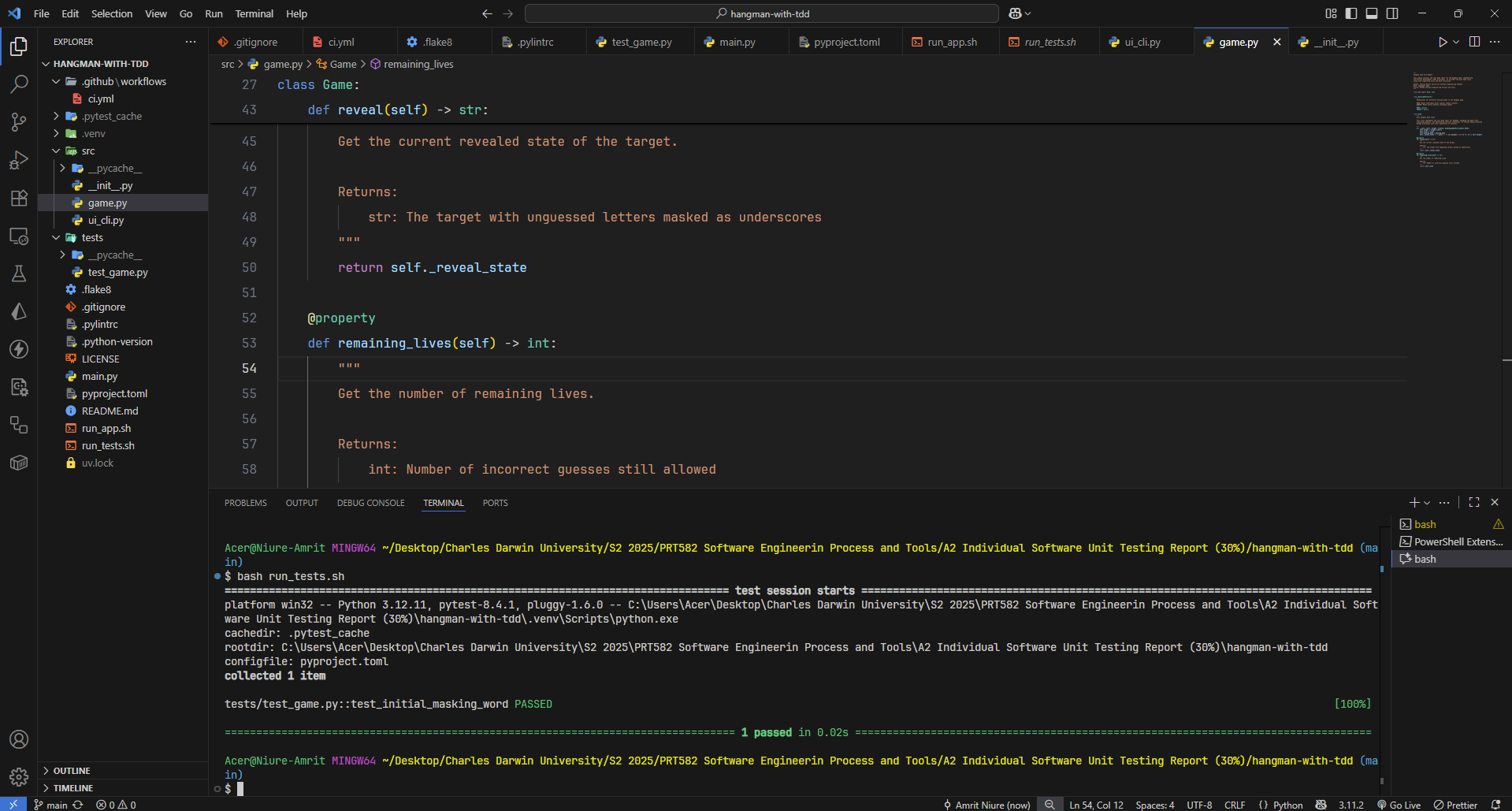
if ch.isalpha():

masked.append("\_")

else:

masked.append(ch)

return "".join(masked)



**Case Insensitive Input**

The main logic of the game is to take the input of any string uppercase or the phrase and process the input by cleaning the input. If a user provides the Capital character, it should take it and turn that input to lower case and process that without any interruption.

Red Phase: The test case for insensitive\_guessing was written first . The test expects the insensitive character and that should not break the game and process it as normal input.

def test\_case\_insensitive\_guessing():

g = Game("Python", lives=3)

res\_lower = g.make\_guess("p")

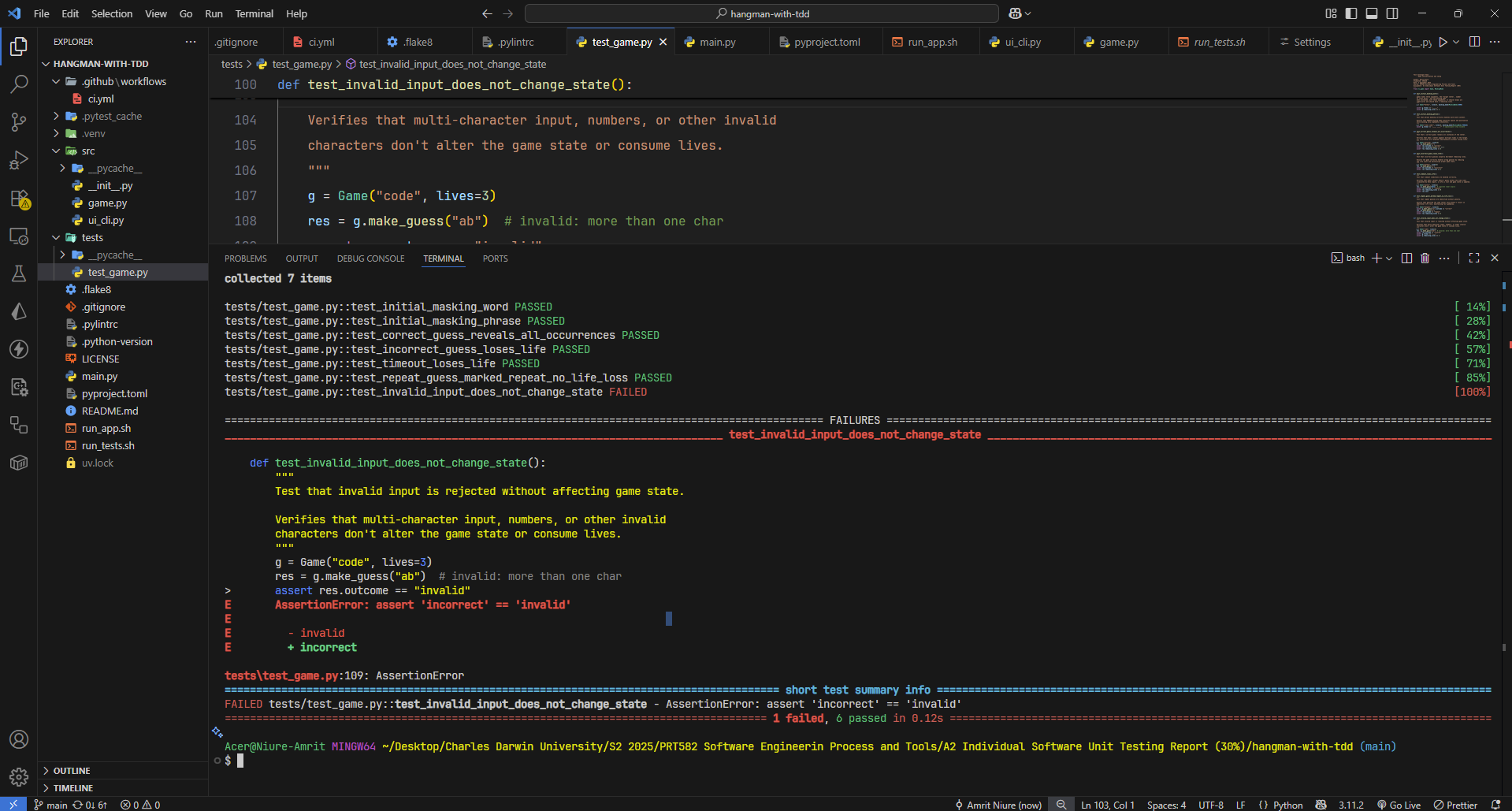
assert res\_lower.outcome == "correct" # is should be processed normal

# Reset for uppercase test

g2 = Game("Python", lives=3)

res\_upper = g2.make\_guess("P")

assert res\_upper.outcome == "correct" # this should also taken as normal input. The make guess method should process the capital P and make it lower case.



After that the logic to validate the input was added that prevented the crashing of the game.

A screenshot of a computer

AI-generated content may be incorrect.

**15-Second Timer**

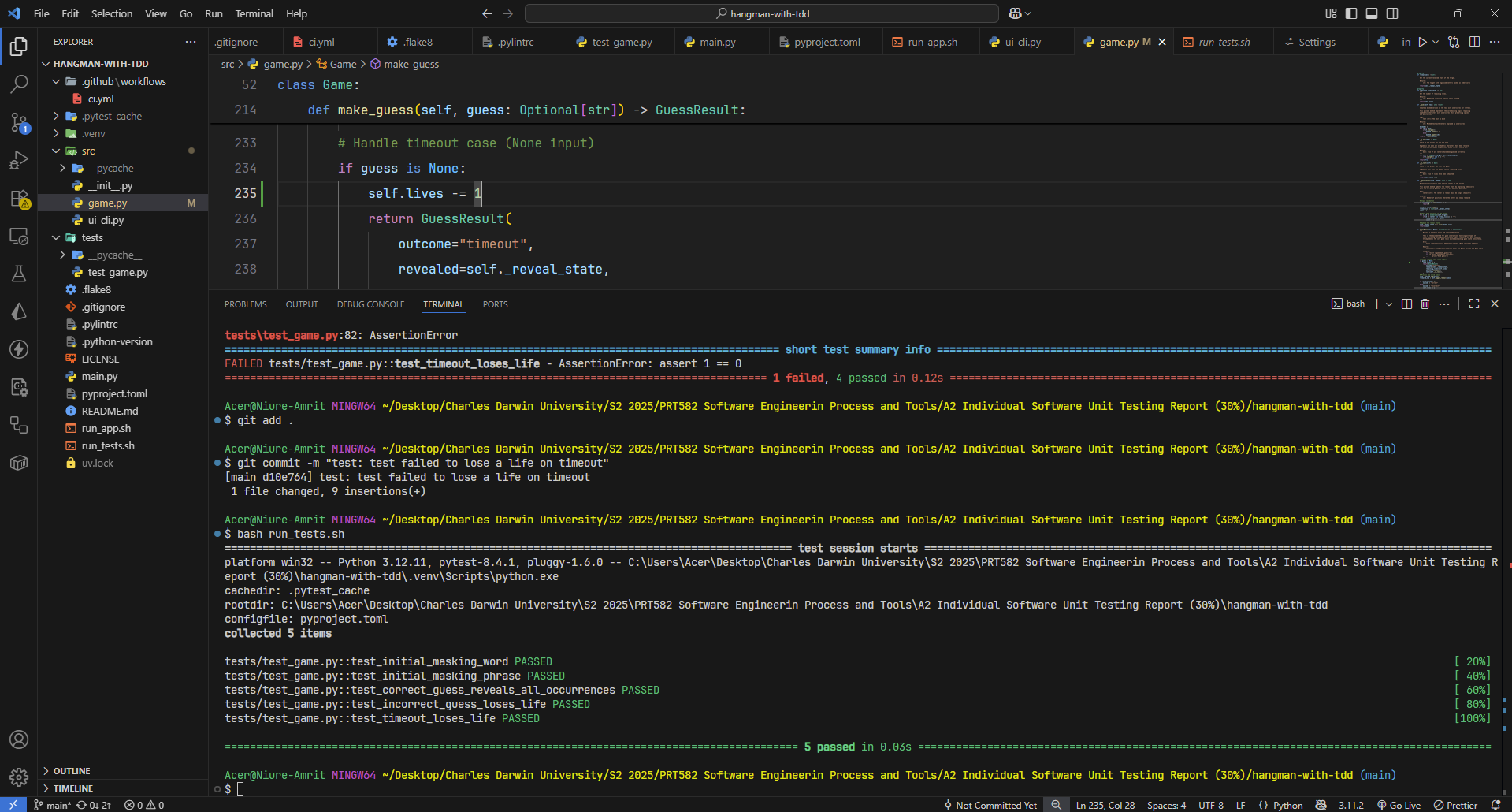
The test\_timeout\_loses\_life was created to ensure the game engine could handle timeouts, simulated by passing None to the guess function.

Red Phase: This test failed (commit d10e764) because the method make\_guess()did not yet have logic to process and handle a None input.

A screenshot of a computer program

AI-generated content may be incorrect.

Greeen Phase: After the test failed, then the implementation of actual function that returns the expected result started. I Implemented the TimedInput Class to fix the problem that does not crashes the game even after the time but loses the life by one.



**Letter Revelation**

The test\_correct\_guess\_reveals\_all\_occurrences was implemented to make sure that all correctly guessed letter are shown after a correct guess. .

**Red Phase:** The test failed at first because the implementation only revealed the position on the first occurrence of the guessed letter.

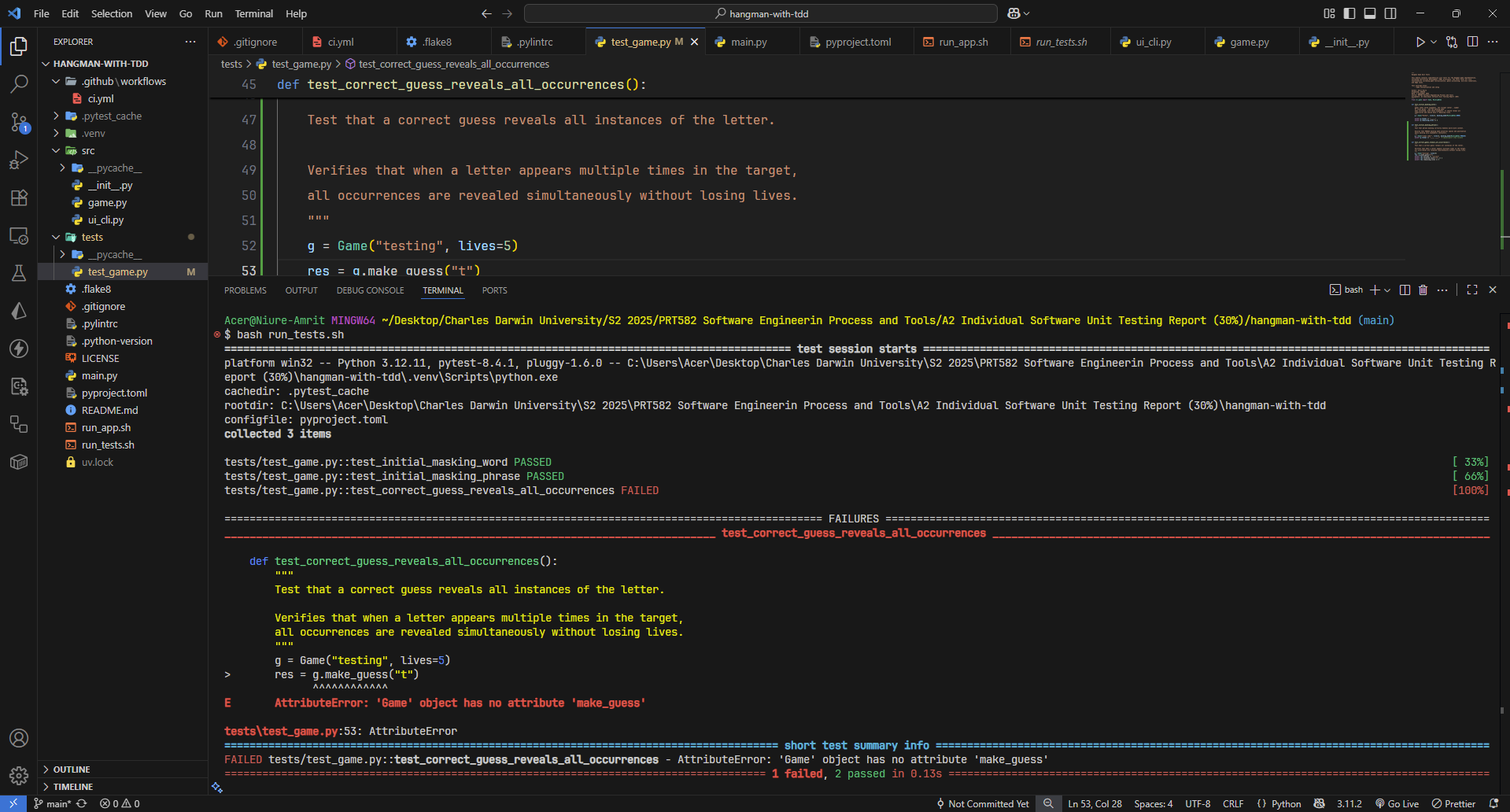
test\_correct\_guess\_reveals\_all\_occurrences

def test\_correct\_guess\_reveals\_all\_occurrences():

g = Game("testing", lives=5)

res = g.make\_guess("t")

assert res.revealed.count("t") == 2



**Green Phase:** The private helper method \_apply\_reveal() was created. This method loops the guessed letter through the target letter, and then updates all the all matching character positions in the \_reveal\_state list, that makes sures all the letters are revealed.

A screenshot of a computer

AI-generated content may be incorrect.

**Life Deduction on Wrong Guess**

The test\_incorrect\_guess\_loses\_life was implemented to check that the life of player is reduced only for an incorrect guess.

**Red Phase:** This test failed (commit dd3e8dc) as the life-deduction logic was not yet implemented.

test\_incorrect\_guess\_loses\_life

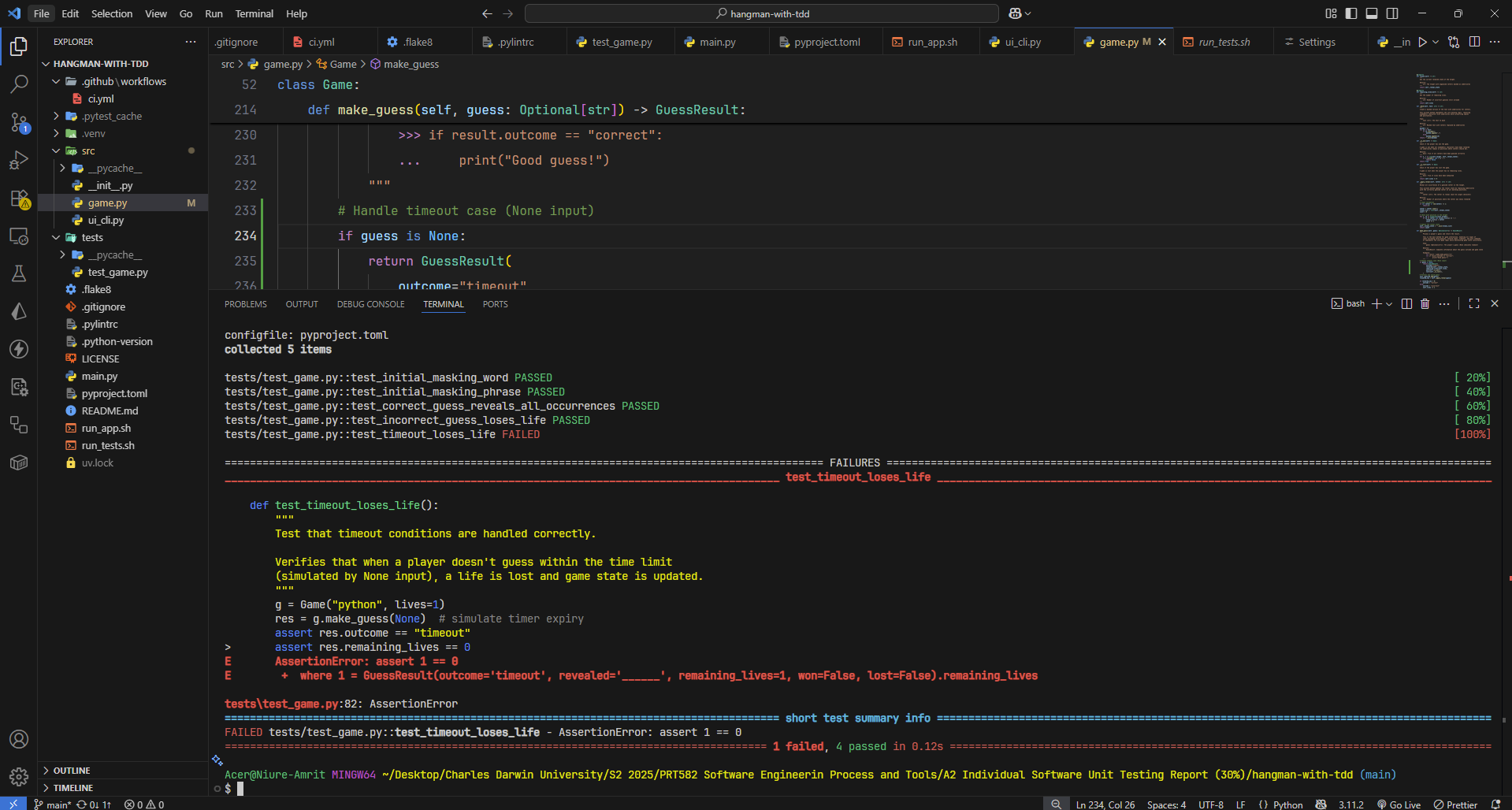
def test\_incorrect\_guess\_loses\_life():

g = Game("python", lives=2)

res = g.make\_guess("z")

assert res.outcome == "incorrect"

assert res.remaining\_lives == 1



**Green Phase:** In the make\_guess() method, after determining a guess is not correct, the line self.lives -= 1 was added. This simple change made the test pass.

A screenshot of a computer program

AI-generated content may be incorrect.

**Win/Loss Conditions**

The tests test\_win\_condition and test\_lose\_condition were implemented to define the games states where an terminating decision is going to be made.

Red Phase: These tests (commits c3c5373, c0dedff) failed because the is\_finisheed method was still not implemented on the game logic and there was no handler for that situation.

Test Cases: test\_win\_condition and test\_lose\_condition

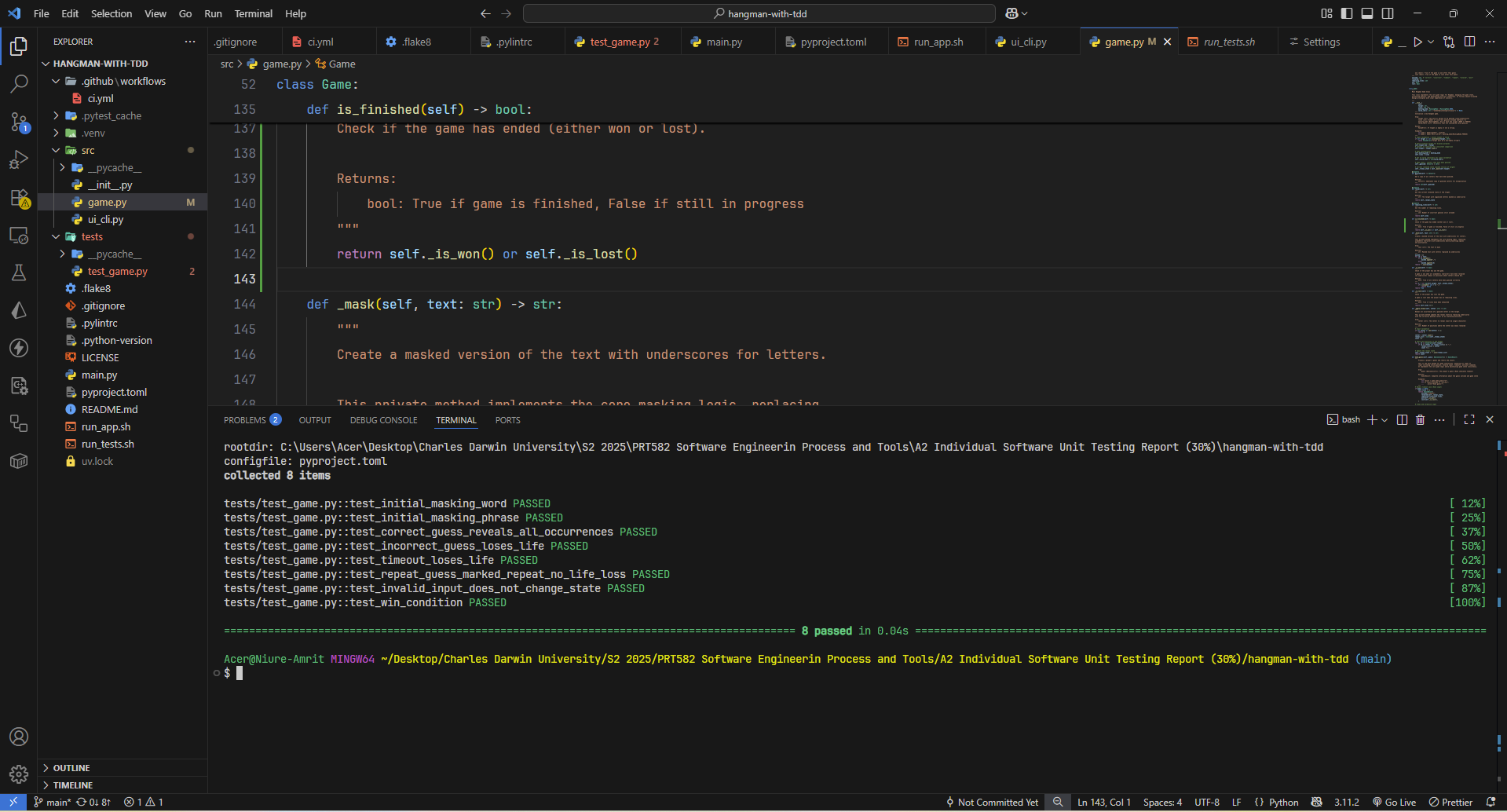
def test\_win\_condition():

g = Game("aa", lives=5)

g.make\_guess("a")

assert g.is\_finished()

**Green Phase:** The private helper methods \_is\_won, this method checks for remaining underscores and \_is\_lost(), which checks if the life of the user is greater than zero were implemented. The public is\_finished() method was then created to call both is\_won and is\_lost method that allowed to pass the test.



**Architectural Design and Code Quality**

The Test Driven Development process directly influenced industry standard high quality architecture of the final game.py module.

* Separation of Concerns: The game class only handles the game logic, the state of the game and its rules. It has no knowledge of of the user interface like printing and showing the ui to the users. So it is independent of any dependencies.
* Immutable Data Transfer: Instead of allowing the external code to get the multiple properties of the game object after a guess on multiple outcomes, the make\_guess method returns a single Data Class which is immutable and makes consistent output format whatever the result and outcome.
* Encapsulation: The logics and the states are encapsulated form the main thread to make it more readable and reusable . If the private methods are needed it is provided by \* The internal state of the game is kept private for example \_guessed and \_reveal\_state . Public access to other methods is provided through read-only @property methods (reveal, remaining\_lives). This prevents external code from accidentally corrupting the game's state.

Lessons Learned:

1. Test Driven Development Methodology Benefits:

* Improved Code Quality: With Test Driven Development, the code became cleaner and manageable with modular code design that improved the quality of the code.
* Better Requirements Understanding: When we already know what should be the result or outcome of our feature, we can understand the problem and implement it better.
* Faster Debugging: Debugging the code is much easier because we now can understand the problem and the workflow of a certain feature beforehand.

1. Testing Best Practices:

* Clear Test Documentation: The test first approach leds the documentation of the test more clear and well named test cases makes the documentation easier to understand.
* Edge case Importance: Testing boundary conditions and other edge cases of the program is identified beforehand and can be solved easily.
* Test Isolation: The test on themselves serves different purposes by each tests, being independent of other test cases.

**Conclusion**

This project successfully demonstrated the effectiveness and the sheer power of Test-Driven Development in building a reliable, well-structured application in a fast paced environment. The development process was followed by clear, executable specifications, by writing tests before implementation, that results in higher code quality and more confidence in the final deliverable.