A white rectangular sign with blue text

AI-generated content may be incorrect.

**SITSE2 - MASTER OF INFORMATION TECHNOLOGY (SOFTWARE ENGINEERING)**

**S225 PRT582 SOFTWARE ENGINEERING: PROCESS AND TOOLS**

[**Software Unit Testing Report**](https://online.cdu.edu.au/ultra/courses/_63651_1/outline/assessment/_6474095_1/overview?courseId=_63651_1)

**Submitted By**

|  |  |  |
| --- | --- | --- |
| **Name** | **Student ID** | **Email Address** |
| Axl Espiritu | S387016 | S387016@students.cdu.edu.au |

**Submitted To Charles Yeo**

CHARLES DARWIN UNIVERSITY – DARWIN CAMPUS

FACULTY OF SCIENCE AND TECHNOLOGY

05 September 2025

**→INTRODUCTION**

The purpose of this assignment was to develop a Hangman game in Python using a Test-Driven Development (TDD) approach. Hangman is a classic word-guessing game where the player must identify a hidden word or phrase within a limited number of attempts.

Python was chosen because it is simple, readable, and widely used in teaching software engineering practices. For automated unit testing, pytest was used because it is lightweight, widely adopted, and integrates easily with Python projects. To ensure coding style and quality, flake8 and pylint were also applied, since they check formatting, unused code, and overall readability, which are important for writing sustainable software.

**→PROCESS**

The development process strictly followed the TDD cycle: write tests → run tests (fail) → implement code → re-run tests (pass). At each stage, unit tests guided the implementation of program requirements, and screenshots were captured as evidence.

1. **Initial Tests and Setup**

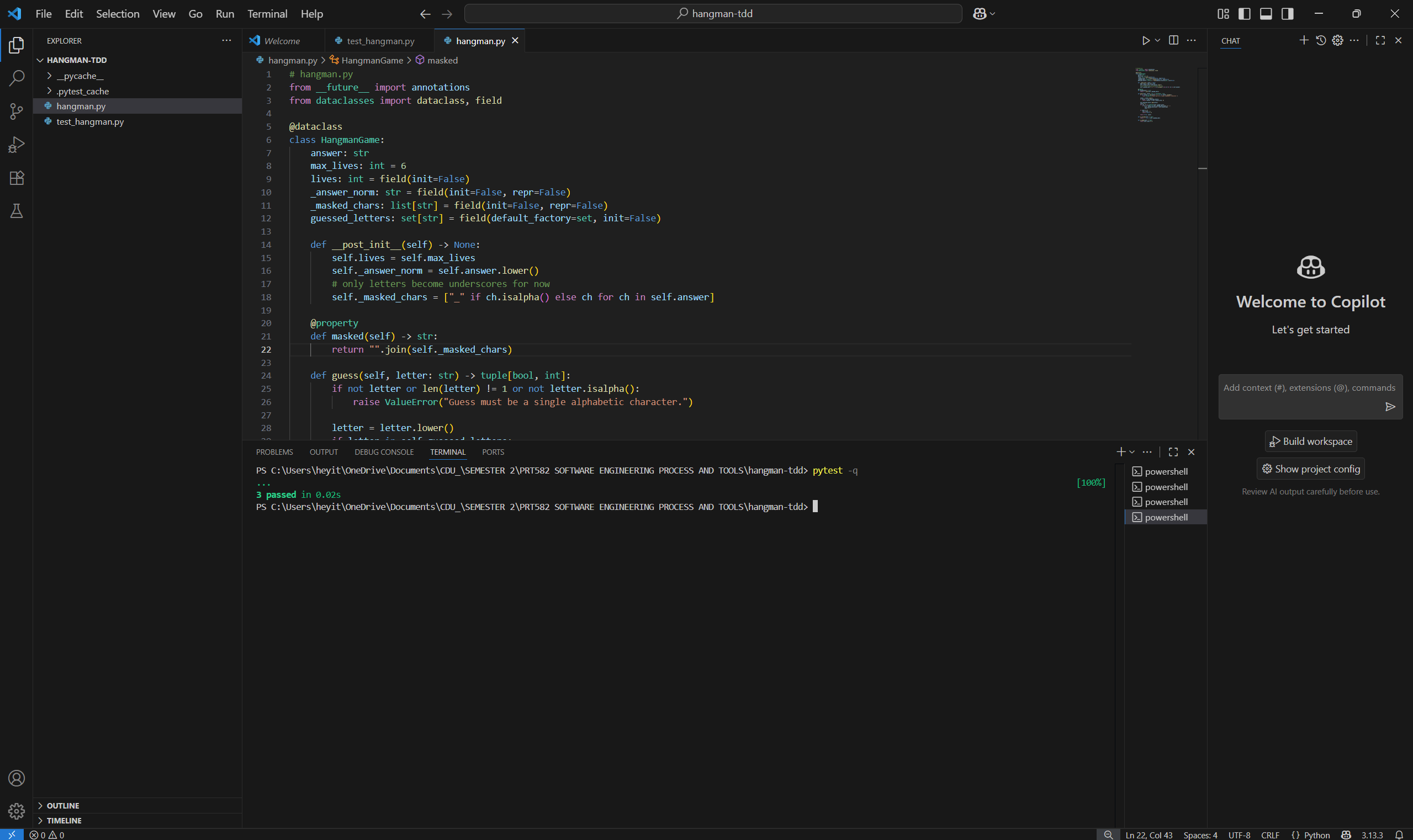
* First, unit tests were written for the Hangman game before any implementation. Running pytest failed because the game file was not yet created.

A screenshot of a computer

AI-generated content may be incorrect.

1. **Basic Game Logic**

* Implemented a minimal HangmanGame class. Tests for showing underscores and correct guesses passed.



1. **Handling Wrong Guesses**

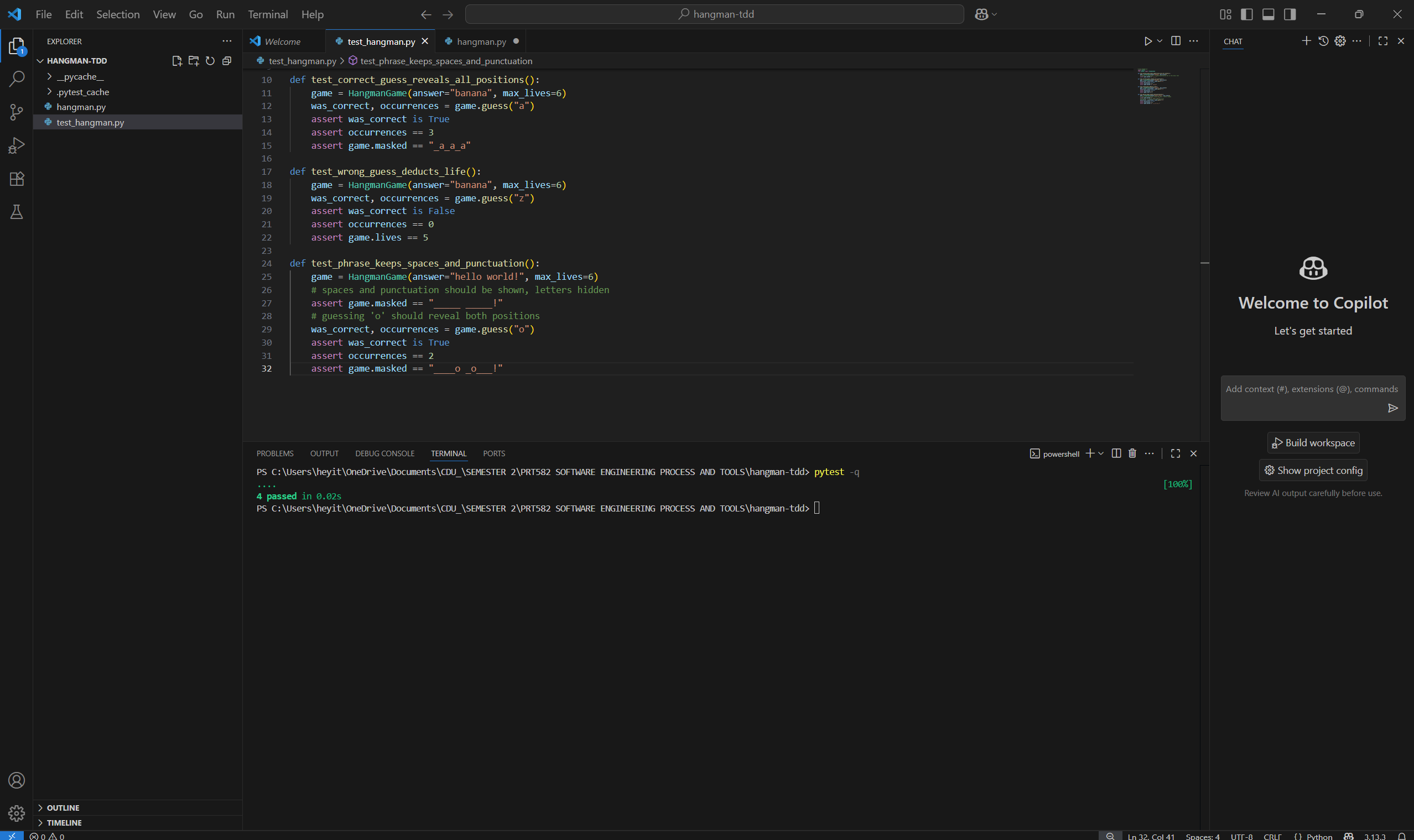
* Added a test to ensure wrong guesses deduct lives. Updated the game logic until the test passed.

A screenshot of a computer

AI-generated content may be incorrect.

1. **Phrase Support**

* Added tests for intermediate mode (phrases). Spaces and punctuation remained visible while letters were hidden.



1. **Random Word/Phrase Providers**

* Implemented random selection functions for both words and phrases. Confirmed with unit tests.

A screen shot of a computer

AI-generated content may be incorrect.

1. **Static Analysis (flake8 and pylint)**

* Ran flake8 and pylint to check code quality. Initial runs showed long-line and formatting issues, which were corrected.

A screenshot of a computer screen

AI-generated content may be incorrect.

1. **Timer Implementation**

* Added a 15-second timer per turn. Initial tests failed, then passed once timeout handling was implemented.

A screenshot of a computer program

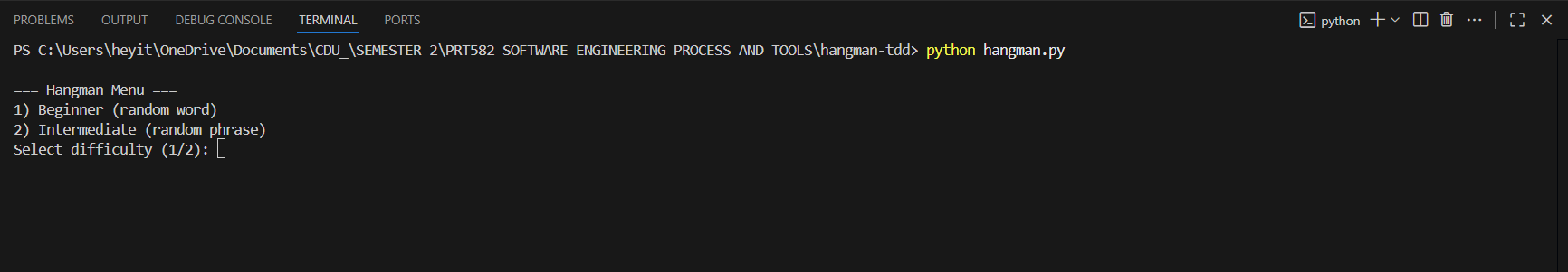
AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

1. **Full Game Loop and Menu**

* Built menus for Beginner and Intermediate modes. Implemented input validation, repeated-letter checks, and play-again prompt.
* Verified game interaction with multiple test runs.



A black screen with colorful text

AI-generated content may be incorrect.

A black screen with a black background

AI-generated content may be incorrect.

A black screen with colorful text

AI-generated content may be incorrect.

A black screen with white text

AI-generated content may be incorrect.

A black screen with blue and white text

AI-generated content may be incorrect.

A black screen with blue and white text

AI-generated content may be incorrect.

A black background with a black border

AI-generated content may be incorrect.

1. **Final Linting Check**

* After corrections, flake8 showed no issues and pylint gave a score above 9/10, which satisfied the rubric.

A computer screen shot of a building

AI-generated content may be incorrect.

**→CONCLUSION**

This project demonstrated how TDD helps ensure that every feature of a program is implemented only after a corresponding test is written. It provided a clear development structure and reduced errors by continuously validating the program against automated tests.

**Lessons learned:**

* TDD makes development incremental and reliable.
* Automated unit testing builds confidence when adding features.
* Using flake8 and pylint enforces clean, professional coding practices.
* Python’s simplicity makes it well-suited for both rapid development and testing.

Overall, this project highlighted the importance of combining **testing** and **linting tools** to produce robust and maintainable software.

**GitHub Repository:**  
*https://github.com/eyeksel/HANGMAN\_TDD*