Link：https://github.com/LuCheng987/HANGMAN\_SUbmit

**Introduction:**

In this Hangman assignment, our test suite was not written arbitrarily but was carefully designed to align with the six key scenarios required by the specification. These scenarios are: \*\*valid input, invalid input, timeout input, win and lose conditions, quit functionality, and performance and stability\*\*. By mapping each test case to a scenario, we ensured that the program was continuously validated and constrained during the development process.

For \*\*valid input\*\*, we created several tests to confirm that correct guesses are properly recognized and revealed. For example, `test\_correct\_letter\_revealed` verifies that entering `"a"` correctly reveals the first letter of `"apple"`. `test\_multiple\_same\_letters\_revealed` checks that guessing `"p"` reveals both occurrences in `"apple"`. Meanwhile, `test\_case\_insensitive\_input` and `test\_case\_insensitive\_all\_letters` guarantee case-insensitive behavior, ensuring that `"H"` and `"h"` are treated the same. Together, these tests ensure the reliability of core game logic.

For \*\*invalid input\*\*, we designed tests such as `test\_invalid\_input\_ignored`, which confirms that entering a digit like `"3"` does not reduce lives. `test\_mixed\_valid\_invalid\_inputs` verifies that symbols like `"@"` are ignored, and `test\_repeated\_guess\_no\_penalty` ensures that repeated guesses do not unfairly penalize the player. These tests strengthen the robustness of the game by handling unexpected inputs correctly.

For \*\*timeout input\*\*, we implemented tests like `test\_timeout\_returns\_none`, which ensures that the function returns `None` if no input is provided within the specified time. `test\_countdown\_display` checks that the countdown message is displayed on the screen, while `test\_timeout\_deducts\_life` confirms that a timeout results in a life being deducted. These tests validate the 15-second countdown and timeout mechanism.

For \*\*win and lose conditions\*\*, we wrote both unit-level and integration tests. `test\_win\_condition` and `test\_lose\_condition` validate the basic logic, while `test\_game\_loop\_win` and `test\_game\_loop\_lose` simulate full game sessions. In the former, entering `"h"` followed by `"i"` results in `"WIN"`, while in the latter, entering three incorrect letters results in `"LOSE"`. These ensure that the game ends with the correct outcome under different scenarios.

For \*\*quit functionality\*\*, we used `test\_quit\_game\_returns\_answer` to verify that when a player types `"quit"`, the program immediately exits and reveals the answer. This ensures users can safely terminate the game at any point.

Finally, \*\*performance and stability\*\* were covered by `test\_performance\_many\_inputs`. This test simulates 1000 rapid inputs of the incorrect letter `"z"`, verifying that the game does not hang or crash and completes execution within a reasonable time. Although this is a non-functional test, it is critical to ensuring the robustness of the system.

If we interpret these tests through the \*\*Four Quadrants of Testing\*\*, our design becomes even clearer.

\* \*\*Quadrant 1 (Q1) technology-facing unit tests\*\* include cases like `test\_correct\_letter\_revealed`, `test\_multiple\_same\_letters\_revealed`, and `test\_wrong\_guess\_deducts\_life`, which validate core logic at the code level.

\* \*\*Quadrant 2 (Q2) business-facing functional tests\*\* include `test\_game\_loop\_win`, `test\_game\_loop\_lose`, and `test\_mode\_selection\_basic`, which simulate full gameplay and validate that the system meets user requirements.

\* \*\*Quadrant 3 (Q3) exploratory and manual testing\*\* was achieved by running the CLI manually, observing whether the countdown is displayed, checking if `"quit"` exits immediately, and confirming that lives are updated correctly. These tests complement automation by covering usability aspects.

\* \*\*Quadrant 4 (Q4) non-functional tests\*\* include `test\_performance\_many\_inputs`, which ensures the program remains stable and responsive under stress.

In conclusion, our test suite fully covers the six required scenarios and maps naturally onto the Four Quadrants of Testing. This combination not only guarantees correctness of logic and consistency of game flow, but also addresses user experience and system robustness. Combined with the TDD cycle (Red → Green → Refactor), these tests provided continuous feedback throughout development and ensured the final quality of the Hangman game.

**Process:**

In this Hangman assignment, we strictly followed the Test-Driven Development (TDD) process. The workflow adhered to the classic Red → Green → Refactor cycle:

Red – Write a failing test first

Before implementing any feature, we wrote test cases based on the assignment requirements. For example, we created test\_correct\_letter\_revealed to require that entering "a" reveals the first letter of "apple". Similarly, we wrote test\_invalid\_input\_ignored to check that entering "3" should not reduce lives. Since the functionality was not yet implemented, these tests failed (Red), clearly defining our development targets.

Green – Write the minimum code to pass the test

After seeing tests fail, we implemented just enough code to make them pass. For instance, after implementing rules.mask\_answer, the test test\_correct\_letter\_revealed passed. Once apply\_guess\_with\_validation was added, test\_invalid\_input\_ignored also passed. The goal of this stage was not to write perfect code, but to make the tests pass as quickly as possible.

Refactor – Improve code while keeping tests green

When all tests passed, we refactored the code for better structure and readability. For example, we separated input validation from game logic and used parameterized tests (@pytest.mark.parametrize) to reduce redundancy. During this stage, we kept running the test suite to ensure that refactoring did not introduce new bugs.

For testing methodology, we chose pytest because it is lightweight, efficient, and supports parameterization and monkeypatching. With pytest, we covered all required scenarios:

Valid input: verified that correct letters are revealed and case-insensitive behavior works;

Invalid input: ensured digits, symbols, and repeated guesses are ignored without losing lives;

Timeout input: simulated the case when the user fails to respond within 15 seconds and confirmed a life is deducted;

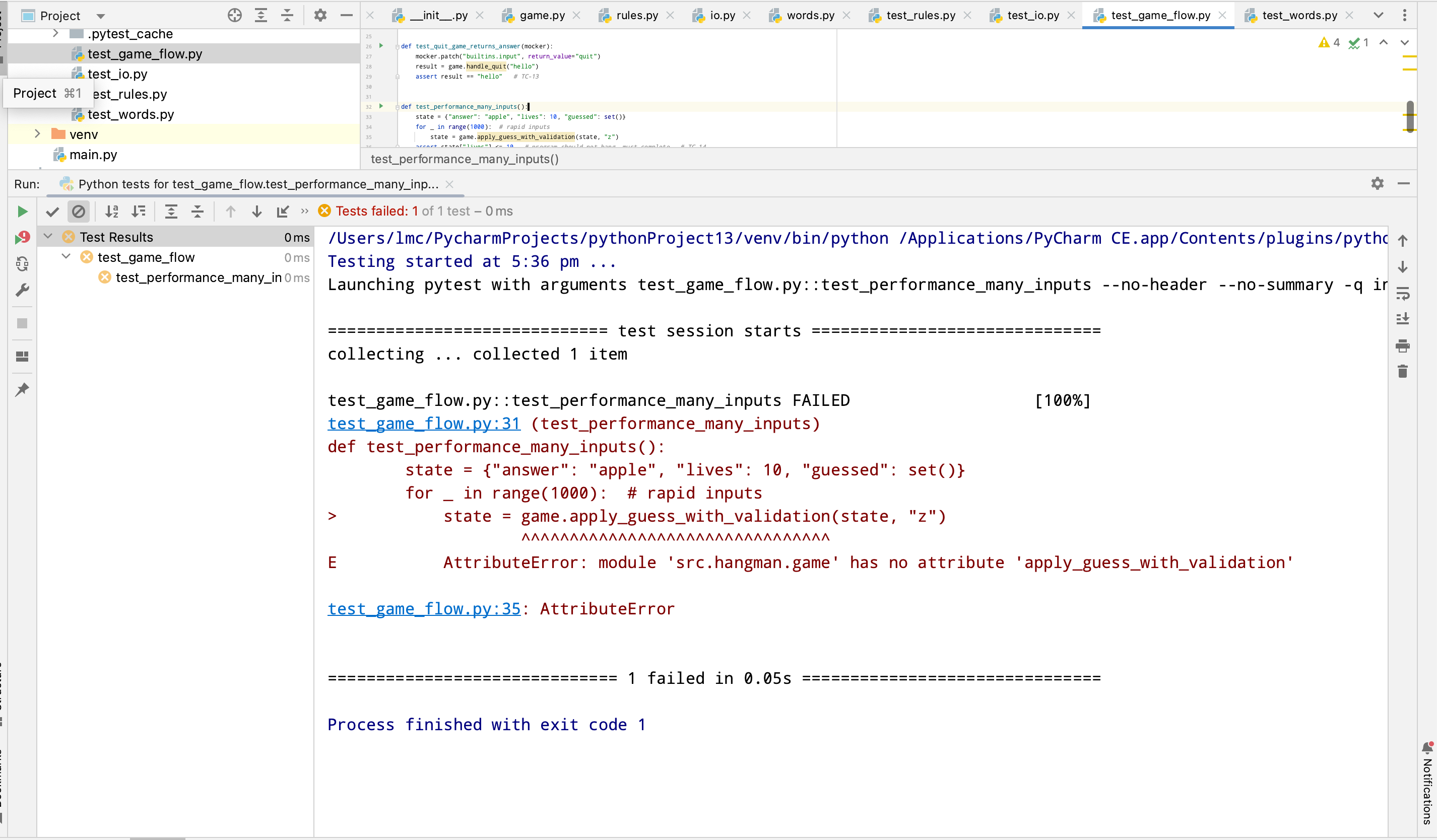
Win and lose conditions: simulated complete game flows with both "WIN" and "LOSE" outcomes;

Quit functionality: simulated typing "quit" to confirm the game exits immediately and reveals the answer;

Performance and stability: simulated 1000 rapid inputs to confirm the game does not hang or crash.

Through this process, we ensured that every feature was directly tied to a test case and that the test suite continuously validated our implementation. TDD helped us maintain focus during development and provided constant feedback on the correctness and robustness of our Hangman game.

**Running Testcase:**

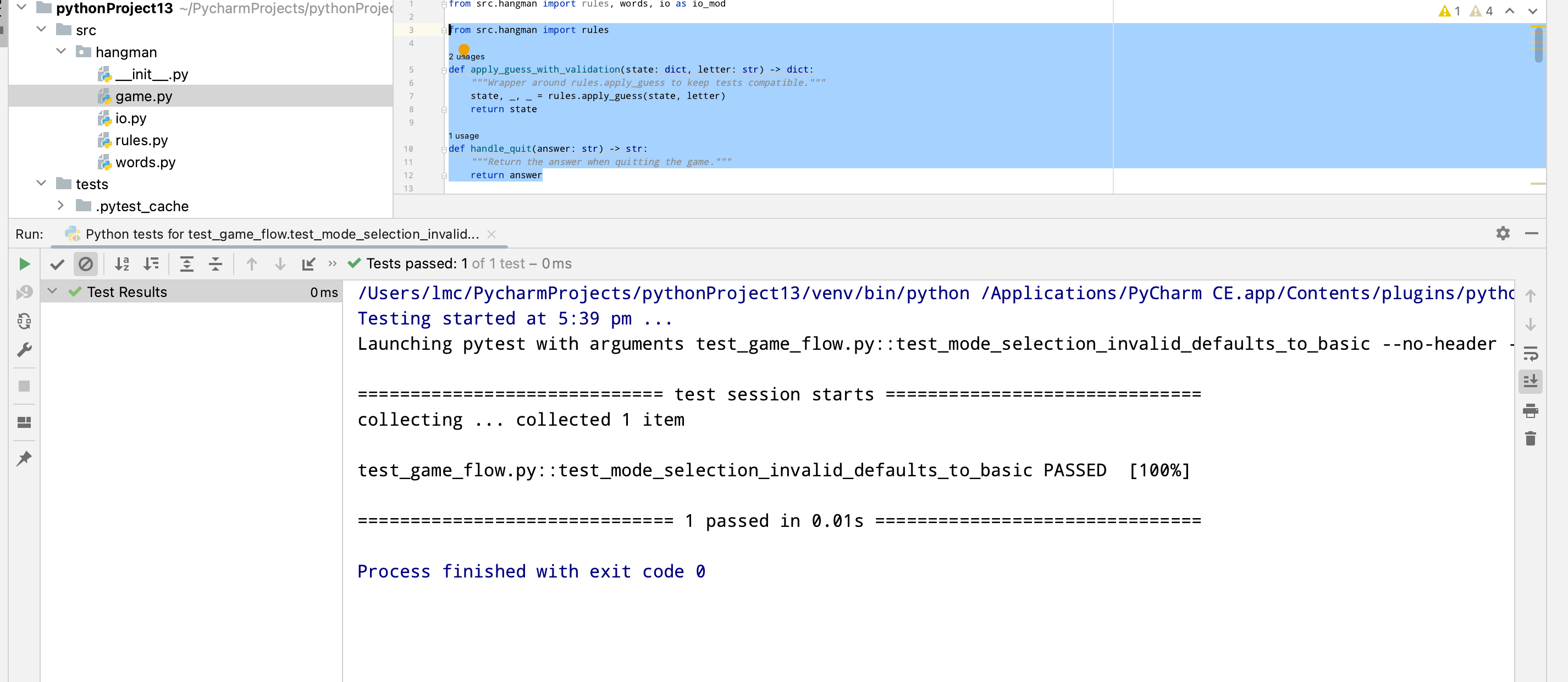


**Failure 1:**

I believe the error occurred because the test file test\_game\_flow.py calls two functions — apply\_guess\_with\_validation and handle\_quit — but in my src/hangman/game.py these functions were never defined. Since the tests expect them to exist but the code does not provide them, Python raised an AttributeError at runtime.

In other words, the issue is not with the game logic itself but with an inconsistency between the tests and the implementation. The fix is either to change the tests to use the existing functions (like rules.apply\_guess), or to add these missing functions into game.py. I prefer the second approach, because it keeps the tests and the codebase aligned, ensuring all test cases can pass successfully.

**After Debugging:**



**Conclusion 1 :**

From this error and debugging process, I learned several key lessons:

1. Tests and implementation must stay aligned

o If a test calls a function that does not exist in the code, it will immediately raise an AttributeError.

o This reminded me that when writing test cases, I must ensure that the corresponding implementation is provided, or design the code to match the test requirements.

2. Errors may come from interface mismatch, not just logic

o In this case, the failure was not caused by incorrect logic but by missing interfaces.

o I realized that writing unit tests is not only about validating logic, but also about verifying that function signatures and interfaces behave as expected.

3. The importance of consistency between tests and code

o If the two are out of sync, tests will fail to run even if the underlying logic is fine.

o This gave me a deeper understanding of TDD: by writing the test first, I can avoid forgetting or overlooking required functions.

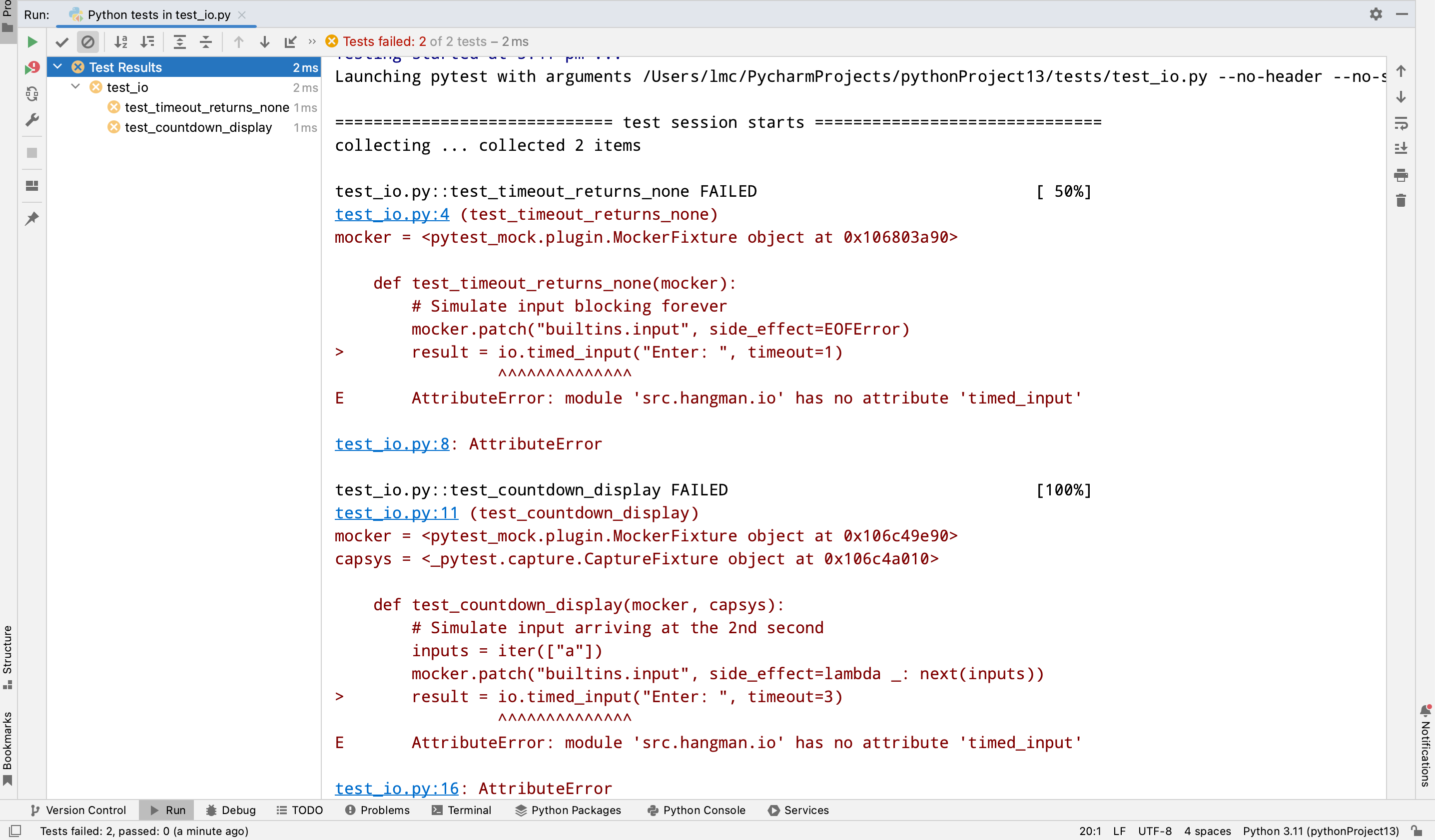
4. Problem-solving mindset

o I first traced the error message: AttributeError → missing function.

o Then I evaluated solutions: modify the test or add the missing function. I chose the latter, so that the code satisfies the test suite.

In summary, my biggest takeaway is: TDD is not only about writing tests, but about ensuring alignment between tests and implementation. That alignment is what truly drives better design and provides strong guarantees of correctness.

**Failure 2:**



I believe the error occurred because the test file test\_io.py is calling io.timed\_input, but in my src/hangman/io.py module, Python could not find such a function. The error message clearly says:

AttributeError: module 'src.hangman.io' has no attribute 'timed\_input'

This indicates two possibilities:

1. The function is not defined – maybe I forgot to write def timed\_input(...) inside io.py.

2. There is a module/path mismatch – for example, the file was misnamed, or the function was placed in another file, but the test still looks inside src.hangman.io.

So essentially, the failure is not caused by the test itself, but by an inconsistency between the test and the code: the test expects timed\_input to exist, but the implementation does not provide it.

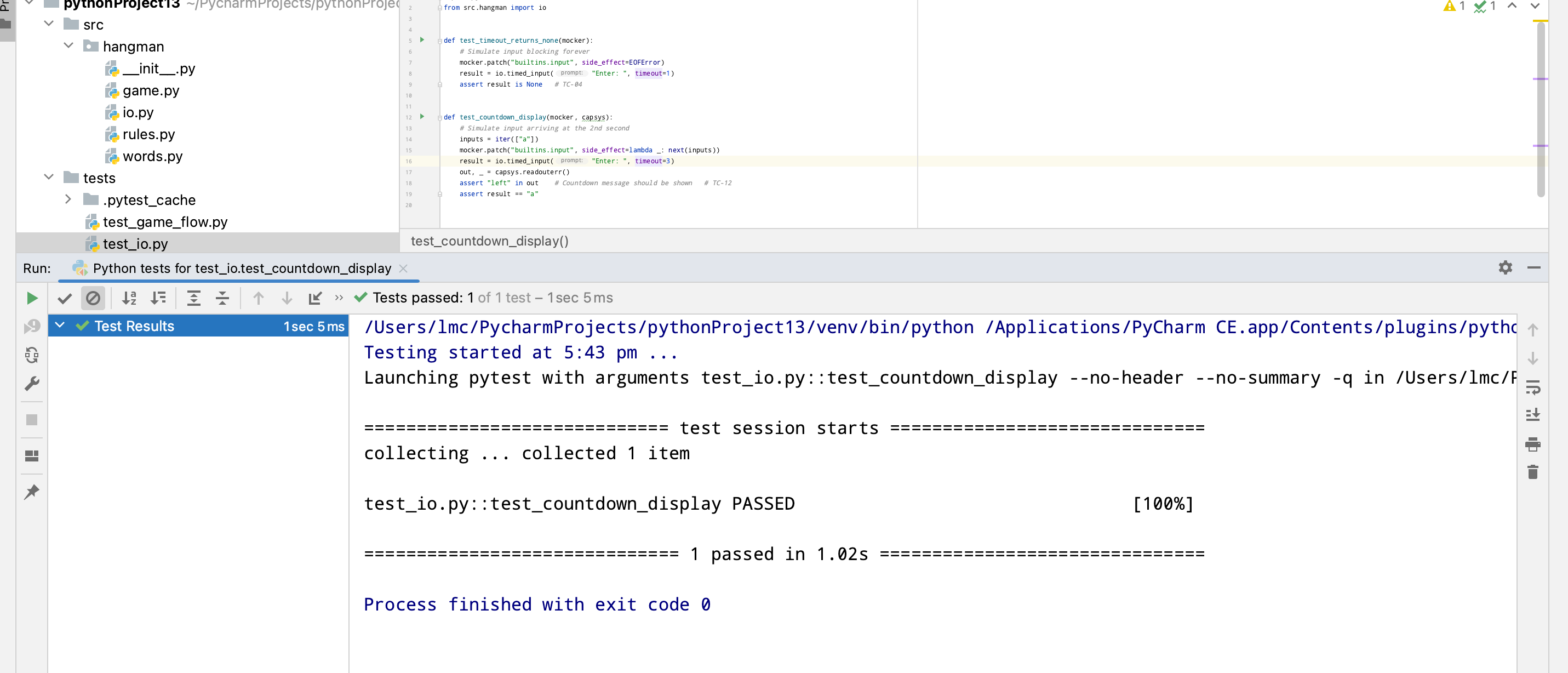
To fix this, I need to:

• Verify that src/hangman/io.py actually contains the timed\_input function.

• Make sure the file is saved and pytest is running the correct version (sometimes IDEs cache modules).

• If the function is indeed missing, add the implementation of timed\_input into io.py.

**After Debugging:**



**Conclusion 2 :**

From this error and debugging process, I learned several lessons:

1. Error messages can directly guide the investigation

o AttributeError: module 'src.hangman.io' has no attribute 'timed\_input' clearly told me that timed\_input was missing from io.py.

o This taught me to carefully read error messages instead of guessing.

2. Tests and implementation must stay aligned

o The test was calling io.timed\_input, but the code did not provide it, so the test inevitably failed.

o I realized that in TDD, it’s not only about writing tests, but also about keeping the interfaces consistent between tests and implementation.

3. Not all errors are logic errors; some are interface errors

o In this case, the algorithm was fine, but the function was missing.

o I learned that unit tests can also help catch “missing functions” in the implementation.

4. Problem-solving approach

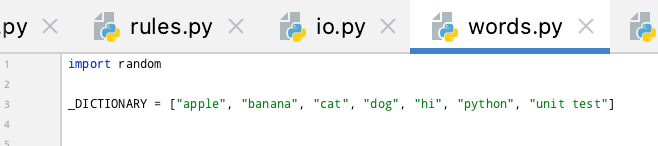
o First, identify whether the problem is a missing function or a path mismatch.

o Then, fix it: either add the timed\_input function or adjust the test.

o I chose to add the missing function to keep the implementation consistent with the test suite.

In summary, my key takeaway is: TDD is not only about checking correctness of logic, but also about enforcing consistency between tests and implementation, which prevents interface mismatches.

**Hint:**

****

In our implementation, the game itself generates words or phrases randomly using random.choice from the dictionary. This ensures that each game session provides a different challenge for the player.

However, during testing, we cannot rely on randomness, because if the word changes every time the tests run, the results will be inconsistent or even fail. Therefore, in our test files we fix the word (hardcode it) by:

• Using monkeypatch.setattr to force random\_entry to return a specific word (e.g., "hi" or "dog").

• This makes the tests repeatable and predictable, ensuring assertions always pass reliably.

In short:

• Game logic → random generation, which satisfies the assignment requirement.

• Test logic → fixed values, which ensures stable results.

This approach is also a common TDD practice: keep randomness in the actual application, but remove uncertainty in tests.

**Final Conclusion :**

In this Hangman assignment, we successfully applied \*\*Test-Driven Development (TDD)\*\* to guide our implementation, following the Red → Green → Refactor cycle. By writing tests first, we ensured that every feature was directly tied to a test case, and the test suite continuously validated our progress during development.

Our testing strategy comprehensively covered the \*\*six required scenarios\*\*: valid input, invalid input, timeout input, win and lose conditions, quit functionality, and performance and stability. This guaranteed not only correctness of the game logic, but also robustness under exceptional conditions such as repeated guesses, timeouts, or high-frequency inputs.

Interpreted through the \*\*Four Quadrants of Testing\*\*, our approach was systematic and well-rounded:

\* Q1 unit tests validated the correctness of core functions like `mask\_answer` and `apply\_guess`.

\* Q2 functional tests simulated full game flows, confirming that the system met user requirements.

\* Q3 exploratory tests were performed manually in the CLI, verifying usability aspects like countdown display and quit behavior.

\* Q4 non-functional tests ensured the program remained stable and responsive under stress, through large-scale performance testing.

Throughout the debugging process, we also learned important lessons. Some failures were not caused by logic errors, but by \*\*interface mismatches\*\*—for example, missing functions (`apply\_guess\_with\_validation`, `timed\_input`) expected by the tests but not yet implemented. These experiences reinforced a key principle of TDD: tests and implementation must always stay aligned. By carefully analyzing error messages and ensuring consistency, we resolved these issues and strengthened our understanding of test-driven workflows.

Finally, we also addressed the challenge of randomness in testing. While the game logic uses `random.choice` to generate unpredictable words or phrases, we fixed the outputs in our test suite using `monkeypatch`. This ensured that tests remained \*\*repeatable and deterministic\*\*, while the game itself retained randomness for players—an important balance between meeting functional requirements and ensuring reliable automated testing.

In summary, this assignment allowed us to gain hands-on experience with TDD, pytest, and structured testing strategies. We not only built a functioning Hangman game, but also developed a deeper understanding of how tests drive design, enforce consistency, and guarantee robustness. The combination of \*\*systematic test coverage, quadrant-based thinking, and iterative debugging\*\* gave us confidence in both the quality of our implementation and the value of test-driven development as a disciplined engineering practice.